ZETA

Entwicklungssystem für Text−Adventures

Benutzerhandbuch

Martin Oehm

Inhaltsverzeichnis

Einleitung 1
Über dieses Handbuch 1
Danksagungen 2

Teil I: Die Sprache von ZETA.....3

1 Grundlagen 4
1.1 Erste Schritte 4
1.2 Routinen und Anweisungen 4
1.3 Zahlen und Variablen 5
1.4 Rechenoperationen 9
1.5 Argumente von Routinen und Rückgabewerte 12
1.6 Bedingungen 14
1.7 Schleifen 18
1.8 Felder 21
1.9 Texte 25

2 Objekte und Klassen 31
2.1 Das Konzept der Objekte 31
2.2 Der Objektbaum 31
2.3 Attribute 35
2.4 Eigenschaften von Objekten 36
2.5 Klassen 39
2.6 Verschiedene Datentypen 42
2.7 Mitteilungen an Objekte 43
2.8 Erzeugen und Vernichten von Objekten 46

3 Systemfunktionen und Besonderheiten 48
3.1 Eingabe von Text 48
3.2 Textkanäle 50
3.3 Schrift und Farbe 50
3.4 Der Bildschirm 52
3.5 Laden und Speichern 53

Teil II: Arbeiten mit dem Compiler.....56

4 Der Compiler und seine Optionen 57
4.1 Optionen beim Aufruf des Compilers 57
4.2 Steuerdateien 59
4.3 Steueranweisungen im Quelltext 60

Inhaltsverzeichnis

5 Fehler suchen und beseitigen 67
5.1 Fehler beim Kompilieren 67
5.2 Logische Fehler im Programm 71

Teil III: Die Adventure- Bibliothek.....76

6 Einführung in die Bibliothek 77
6.1 Aller Anfang ... .77
6.2 Interaktion mit Objekten 78
6.3 Aktionen und Befehle 82
6.4 Erweiterungen der Bibliothek 82

7 Das Modell der Spielwelt 83
7.1 Räume und Richtungen 83
7.2 Türen und Verbindungen 85
7.3 Behälter, Ablagen und Unterobjekte 88
7.4 Begehbare Objekte, Sitze und Fahrzeuge 91
7.5 Kleidung 95
7.6 Objekte zum Ein- und Ausschalten 95
7.7 Lesen und nachschlagen 96
7.8 Andere Personen im Spiel 97
7.9 Kommunikation 98
7.10 Dunkelheit 100
7.11 Die Sinne 102
7.12 Der Spieler 103
7.13 Zeitabhängige Vorgänge 107
7.14 Der Anfang und das Ende des Spiels 109
7.15 Die Statuszeile 112

8 Beschreibungen 114
8.1 Namen von Objekten und Räumen 114
8.2 Listen von Objekten 119
8.3 Mehrere gleiche Objekte 123

9 Der Parser 125
9.1 Verben 125
9.2 Verben 126
9.3 Satzmuster 126
9.4 Objekte 126
9.5 Parser-Routinen 126
9.6 Gültigkeit von Objekten 126

Inhaltsverzeichnis

Teil IV: ZETA für Fortgeschrittene 127

10 Weitere Funktionalitäten 128
10.1 Routinen mit beliebig vielen Argumenten 128
10.2 Implizites „jenach“ 129

11 Daten für den Parser 131
11.1 Definition von Objektvokabular 131
11.2 Grammatikdefinitionen 132

Anhang A: Spezifikation der Sprache 137

Anhang B: Überblick über die Bibliothek 157

Lösungen der Aufgaben 167

Index 183

Einleitung

ZETA ist ein Entwicklungssystem für Textadventures, genauer gesagt, für Textadventures in deutscher Sprache. Es basiert auf dem *Z*-Code, der von *Infocom* entworfen wurde und für den es Interpreter auf einer großen Anzahl von Plattformen gibt.

ZETA entstand aus dem Wunsch heraus, ein System zu haben, in dem umfangreiche Adventures auf Deutsch geschrieben werden können. Ich hatte zuvor den Text-Adventure-Generator T.A.G. entwickelt, der zwar einen zuverlässigen Parser und gute Textausgabe bietet, aber einige große Nachteile hat: Der Interpreter ist nur für MS-DOS erhältlich und oft instabil. Außerdem schränken die einfache Syntax und der „fest verdrahtete“ Parser die Möglichkeiten beim Entwerfen von Spielen ein.

Graham Nelsons *Inform*, genau wie ZETA ein Compiler für die virtuelle *Z*-Maschine, ist sehr flexibel und weit entwickelt. Er wird von vielen benutzt und ist wohl das Standard-Werkzeug zum Erstellen von Adventures – zumidest im englischen Sprachraum. Es gibt weiterhin eine ganze Anzahl von Interpretern für das *Z*-Format, das klar spezifiziert ist, so dass man mit *Inform* erzeugte Spiele sowohl auf Großrechnern als auch auf Organizern und Handhelds spielen kann.

Mit *Inform* kann man Adventures in anderen Sprachen außer Englisch schreiben, die aber auf der relativ einfachen englischen Syntax aufbaut und in seinen vorgefertigten Satzmustern Grammatik und Semantik durcheinanderwirft. Die Möglichkeiten zum Anpassen des *Inform*-Parsers sind daher meiner Meinung nach eingeschränkt.

Über dieses Handbuch

ZETA besteht aus dem *Compiler*, der schließlich die Spieldatei erzeugt, und der *Adventure-Bibliothek*, bereits geschriebenem Code, der das Schreiben von eigenen Adventures ermöglicht.

Der erste Teil dieses Buchs befasst sich mit den Grundlagen von ZETA, der Syntax, den Elementen der Spieldatei und dem Compiler. Die folgenden Teile des Handbuchs zeigen dann, wie man ein Adventure schreibt und erklären die Einzelheiten der Bibliothek. Wer sich nicht lange mit den Grundlagen aufhalten will, kann mit Teil III beginnen und Teil I nur dann zu Rate ziehen, wenn etwas unklar ist. Die Syntax von Zeta ist nicht schwer zu verstehen, sie lehnt sich ein wenig an C und Java an, benutzt aber deutsche Schlüsselwörter.

Die Teile des Buchs, die sich mit der Bibliothek beschäftigen, enthalten viele Code-Beispiele, die in Schreibmaschinenschrift gedruckt sind. Um den Leser nicht mit Code zu erschlagen,

werden neben den Beispielen auch Aufgaben gestellt, deren Lösungen den Leser hoffentlich etwas beschäftigen, bevor er sie am Ende des Buchs nachschlägt.

Danksagungen

…

Teil I:

Die Sprache von ZETA

1 Grundlagen

1.1 Erste Schritte

Z-Code ist binärer Code und damit für den Menschen nicht lesbar. ZETA wandelt eine lesbare Beschreibung des Spiels, den so genannten „Quelltext“ in eine binäre Spieldatei im *Z*-Code um. Diese Spieldatei kann dann mit einem Interpreter ausgeführt, das erzeugte Spiel gespielt werden.

Dieses Handbuch beschreibt den Aufbau und die Syntax des Quelltextes. Der Quelltext ist ein ASCII-Dokument, der in jedem Texteditor erstellt werden kann. Bei umfangreicheren Textverarbeitungsprogrammen muss allerdings darauf geachtet werden, dass keine Formatierungen abgespeichert werden. Die Quelldatei darf nur reinen Text enthalten. Es empfiehlt sich, einen Editor zu verwenden, der keine Formatierngen zulässt. Wichtiger ist eine gute Suchfunktion.

1.2 Routinen und Anweisungen

Unser erstes Programm ist noch recht simpel, es soll einfach den Text „Hallo, Welt!“ auf dem Bildschirm ausgeben. Der Quelltext dazu sieht so aus:

```
// Ein erstes Beispiel

Hauptprogramm {
    "Hallo, Welt!";
}
```

Die erste Zeile im Beispiel wird von ZETA ignoriert. Man nennt dies einen *Kommentar*, hier kann der Programmierer Notizen einfügen, um den Überblick im Quelltext zu behalten. Wie in C oder Java, gibt es zwei Arten von Kommentaren. ZETA ignoriert alles, was auf einer Zeile nach einem doppelten Schrägstrich `//` kommt und alles, was zwischen `/*` und `*/` steht. Wenn man die letzte Möglichkeit benutzt, kann der Kommentar über mehere Zeilen gehen.

Danach wird das `Hauptprogramm` definiert. Jedes erstelle Spiel muss eins besitzen, es wird nach Aufrufen der Spieldatei im Interpreter ausgeführt. Wenn alle Anweisungen im Hauptprogramm abgearbeitet sind, hält der Interpreter an.

Die beiden geschweiften Klammern schließen den Ausführungsblock des Hauptprogramms

ein. Momentan enthält dieser Block nur eine Anweisung: Text in doppelten Anführungszeichen. Diese Anweisung bedeutet: Gib diesen Text auf dem Bildschirm aus.

In ZETA müssen alle Befehle mit einem Strichpunkt oder Semikolon abgeschlossen werden, es sei denn, sie stehen vor einer geschweiften Klammer. Das Semikolon in unserem Beispiel wäre also nicht unbedingt nötig gewesen, aber es schadet nicht, es zu setzen. Außerhalb von Anführungszeichen behandelt ZETA Leerzeichen, Tabulatoren und Zeilenumbrüche gleich: Als so genannten *white space*, der zum Abtrennen von einzelnen Wörtern und Symbolen im Quelltext dient.

Folgender Quelltext hätte also dieselbe Spieldatei erzeugt:

```
Hauptprogramm { "Hallo, Welt!" }
```

Neben dem Hauptprogramm gibt es noch weitere Blöcke mit Anweisungen, die ähnlich aufgebaut sind, sie heißen *Routinen*. Eine Routine wird mit der Direktive `Routine` definiert:

```
Routine Hallo {
    "Hallo, Welt!";
}

Hauptprogramm {
    Hallo();
}
```

Jetzt steht ein anderer Befehl im Hauptprogramm: `Hallo()` bedeutet: Führe die Anweisungen, die in der Routine `Hallo` stehen, aus und mache dann mit der nächsten Anweisung weiter. Man sagt, die Routine `Hallo` wird *aufgerufen*. (Die beiden Klammern sind hier wichtig, auch wenn sie leer sind. Nur, wenn sie angegeben werden, weiß ZETA, dass die Routine aufgerufen werden soll.)

Dieses Programm bewirkt dasselbe wie unser erstes Programm, das nur aus dem Hauptprogramm bestand. Routinen sind immer dann nützlich, wenn Anweisungen häufig benötigt werden, in unserem Fall, wenn man oft „Hallo, Welt!“ ausgeben muss.

In anderen Programmiersprachen heißen Routinen auch Subroutinen, Funktionen oder Prozeduren. Es handelt sich dabei aber immer um dasselbe Konzept.

1.3 Zahlen und Variablen

Bis jetzt haben wir nur Texte ausgegeben. Um unser Programm etwas komplexer zu machen, führen wir nun Zahlen ein:

```
Hauptprogramm {
    num(1972);
}
```

Dieses Programm gibt nur die Zahl 1972 aus, mehr nicht. (Die Systemfunktion `num` schreibt einen numerischen Wert.) Zahlen in ZETA bestehen aus einer Folge von arabischen Ziffern, denen ein Minuszeichen vorangehen kann. Sie sind immer ganzzahlig und liegen im Bereich von –32.768 bis 32.767.

Neben dieser Dezimalschreibweise kann man Zahlen auch in Hexadezimal- und Binärschreibweise angeben, dazu stellt man ihnen ein oder zwei Dollarzeichen voran (`$` bzw. `$$`). Dann kann allerdings kein negatives Vorzeichen angegeben werden, negative Zahlen *–x* werden als 65536 – *x* betrachtet.

Folgende Anweisungen sind also völlig gleich und erzeugen denselben *Z*-Code:

```
num(1972);
num($07B4);
num($$11110110100);
```

Im Binärsystem mit der Basis 2 werden nur die Ziffern 0 und 1 benutzt. Das Hexadezimalsystem arbeitet mit der Basis 16, und benötigt Ziffern mit Werten von 0 bis 15. Die „Ziffern“ mit Werten über 9 werden durch die Buchstaben `A` bis `F` bzw. `a` bis `f` dargestellt.

Diese Zahlen sind konstant, `1972` hat immer den Wert 1972. Weitere Konstanten sind z.B. Zeichenketten innerhalb doppelter Anführungszeichen wie `"1972"` oder Vokabeln in Hochkommas `'1972'`, sie haben als Wert die Adresse, an der sie abgespeichert sind. Einzelne Zeichen in Hochkommas haben den Zahlenwert des Zeichens im von ZETA benutzten Zeichensatz: `'z'` entspricht 122.

× Um als Vokabeln einzelne Buchstaben anzugeben, etwa „i“ als Abkürzung für „Inventar“, muss man ihnen einen *Backslash*, einen gegenläufigen Schrägstrich, anhängen: `'i\'`.

. . . . .

Im Gegensatz dazu können *Variablen* ihren Zahlenwert ändern. Je nach dem Bereich ihrer Gültigkeit unterscheidet man zwischen globalen und lokalen Variablen. Globale Variablen sind von überall im Programm her zugänglich, lokale Variablen sind auf eine Routine beschränkt.

Globale Variablen können außerhalb von Routinen mit der Direktive `Variable` definiert werden.

```
Variable x;

Hauptprogramm {
    num(x);
}
```

Hier wurde eine globale Variable mit dem Bezeichner `x` definiert. Da kein Wert angegeben wurde, hat sie zu Beginn den Wert 0. Das Hauptprogramm gibt also „0“ aus. Man kann der Variable aber einen Anfangswert geben:

```
Variable x = -24;
```

Vielleicht ein paar Worte zu den Bezeichnern: Die Bezeichner, manchmal auch IDs (von engl. *identifier*) genannt, müssen eindeutig sein. Es kann nur eine Variable mit dem Namen `x` geben, und wenn es eine Variable mit dem Namen `x` gibt, darf nicht auch eine Routine so heißen.

Außerdem gelten für Bezeichner folgende Regeln: Sie dürfen nur Buchstaben (auch die deutschen Umlaute und ß), Ziffern und den Unterstrich _ enthalten. Sie dürfen maximal 36 Zeichen lang sein, und das erste Zeichen darf keine Ziffer sein, um sie besser von Zahlen unterscheiden zu können. Erlaubte Bezeichner sind zum Beispiel:

```
x1024   großer_Hügel   ___12gelb}
```

Bei Bezeichnern in ZETA wird nicht zwischen Groß- und Kleinschreibung unterschieden. Weiterhin werden die Umlaute gleich wie ihre Kreuzworträtsel-Pendants `ae`, `oe` und `ue` behandelt. Das ß ist äquivalent zu `ss`. Die Bezeichner `Großer_Hügel` und `grosser_hügel` sind also für ZETA gleichwertig.

Variablen kann mit dem Gleich-Zeichen = innerhalb einer Routine ein neuer Wert zugewiesen werden:

```
Hauptprogramm {
    x = 1972;
    num(x);
}
```

Es wird immer 1972 ausgegeben, egal, welchen Wert `x` vorher hatte.

. . . . .

Lokale Variablen können mit der Anweisung `lokal` definiert werden. Diese Anweisung muss die erste in einer Routine sein. Es können bis zu fünfzehn Variablen angegeben werden, die durch Kommas getrennt werden:

```
Routine Farbe {
```

```
    lokal rot, grün;
    lokal blau;

    num(rot), " ", num(grün), " ", num(blau);
}
```

Diese Routine besitzt drei Variablen, `rot`, `grün` und `blau`. Alle drei haben zu Beginn den Wert null. Man kann lokalen genau wie globalen Variablen einen Anfangswert geben. Mehrere `lokal`-Anweisungen können aufeinander folgen, es müssen nur die ersten Anweisungen in jeder Routine sein:

```
Routine {
    lokal x = 12, y = 2*x, z = 2;
    lokal Stadt = "Granada";
    ...
}
```

Lokale Variablen gelten nur innerhalb der Routine, in der sie definiert wurden. Folgendes Beispiel erzeugt einen Fehler:

```
Routine Unbekannt {
    num(a);
}

Hauptprogramm {
    lokal a;
    Unbekannt();
}
```

Die Variable `a` ist innerhalb von `Unbekannt` nicht bekannt. Lokale Variablen können Bezeichner haben, die bereits vergeben wurden, z.B. an eine globale Variable. Dann bezieht sich der Bezeichner immer auf die lokale Variable, auf die globale Variable kann in dieser Routine nicht zugegriffen werden:

```
Variable a = 96;

Routine Zweideutig {
    lokal a = 15;
    num(a), "\n";
}

Hauptprogramm {
    Zweideutig();
    num(a);
}
```

Dieses Programm gibt 15 und 96 aus. (Die Zeichenkombination `\n` in der letzten Zeile von `Zweideutig` bedeutet: Schreibe einen Zeilenumbruch.) Der Wert der globalen Variable `a`

wird durch die Zuweisung in `Zweideutig` nicht beeinträchtigt.

. . . . .

Schließlich gibt es noch die Möglichkeit, selbst Konstanten zu definieren. Dazu benutzt man die Direktive `Konstante`:

```
Konstante meinJahr = 1972;
```

Damit wird im gesamten Programm `meinJahr` als die Zahl 1972 interpretiert. Anweisungen wie `a = meinJahr` und `a = 1972` erzeugen denselben Code. Eine Besonderheit gibt es bei Konstanten: Wenn kein Wert mit = zugewiesen wird, erzeugt ZETA eine Konstante, deren Wert sich von allen anderen ohne Wert definierten Konstanten unterscheidet.

```
Konstante Karo, Herz, Pik, Kreuz;
```

erzeugt vier voneinander verschiedene Konstanten, ohne dass der Programmierer weiß, welchen Wert sie genau haben.

Alle Variablen und Konstanten müssen definiert werden, bevor sie zum ersten Mal in einem Ausdruck verwendet werden!

1.4 Rechenoperationen

Außer dem Zuweisungsoperator = gibt es in ZETA eine Anzahl weiterer, insbesondere arithmetischer Operatoren, so dass Zuweisungen wie

```
Position = 8*Reihe + Spalte;
```

möglich sind. Die arithmetischen Operatoren sind +, -, *, / und % für Addition, Subtraktion, Multiplikation, ganzzahlige Division und Rest einer ganzzahligen Division.

Ob zwischen Operatoren und Bezeichnern Leerzeichen stehen oder nicht, ist egal.

Ganzzahlige Division bedeutet, dass das Ergebnis einer Division immer ganzzahlig ist, in ZETA ist `23/5` gleich 4 und `23%5` gleich 3. Es gilt:

```
a == (a/b)*b + (a%b)
```

× Die Division und die Bildung des Rests sind vorzeichenbehaftet. Das Ergebnis der Division ist negativ, wenn der Divedend `a` und der Divisor `b` verschiedene Vorzeichen haben,

der Rest ist negativ, wenn der Dividend `a` negativ ist:

```
−7/3 == −2        7/−3 == −2        −7/−3 == 2
−7%3 == −1        7%−3 == 1         −7%−3 == −1
```

. . . . .

Das Beispiel oben hätte auch als

```
Position = Spalte + 8*Reihe;
```

geschrieben werden können, die Reihenfolge der Operatoren spielt keine Rolle, das Ergebnis ist dasselbe. In jedem Fall wird `8*Reihe` zuerst ausgewertet. „Punktrechnung geht vor Strichrechnung“: `*`, `/` und `%` binden die Operanden stärker als `+` und `-`.

Um die Bindung der Operatoren zu ändern, kann man runde Klammern setzen:

```
5 * 8 − 12 == 28
5 * (8 − 12) == −20
```

Die arithmetischen Operatoren mit der höchsten Bindung sind die unären Operatoren `+` und `-`. Unär bedeutet, dass sie sich nur auf einen Operanden beziehen, nämlich den Wert rechts von ihnen. `+a` bewirkt nichts, `-a` ändert das Vorzeichen von `a` und ist äquivalent zu `(0 - a)`.

. . . . .

Da die Zahlen in ZETA nur in einem begrenzten Bereich liegen, kann es zu so genannten Überlauffehlern kommen: `1024*100` liegt außerhalb des gültigen Bereichs, die Zahl wird „abgeschnitten“. Darauf muss man besonders bei der Multiplikation achten.

Ein weiterer wichtiger Punkt ist die Division durch null oder die Restbildung einer solchen Division. Sie ist nicht erlaubt und führt während des Programmablaufs zu einem kritischen Fehler, der die Ausführung sofort abbricht. Diese Fehler heißen *Laufzeitfehler*. Daher sollte der Programmierer darauf achten, dass er solche Fälle abfängt.

Neben der Zuweisung mit `=` gibt es die In- und Dekrementoperatoren `++` und `--`, die den Wert einer Variable um eins erhöhen oder vermindern. Stehen diese Operatoren vor der Variable, so wird der Wert erst geändert und dann gelesen, stehen sie dahinter, so wird der Wert erst gelesen und anschließend modifiziert.

```
x = 5;
num(x++), " ", num(++x);
```

Dies schreibt 5 und 7, der Wert von `x` ist danach 7. Diese Operatoren sind etwas gewöhnungsbedürftig und werden unter anderem in Sprachen wie C und Java benutzt. Sie können auch alleine in einer Anweisung stehen, dann ist es egal, ob der Operator vor oder nach einer Variable kommt: `x++` entspricht `x = x+1`.

Eine weitere Abkürzung betrifft das relative Verändern von Variablen. Häufig tauchen Anweisungen wie diese auf:

```
x = x + 2*(y - 1);
```

Hier wird die Variable `x` um den Ausdruck nach dem Pluszeichen erhöht. Dies kann man kurz schreiben als:

```
x += 2*(y - 1);
```

Anstelle des `+` kann hier auch ein anderer arithmetischer Operator stehen, erlaubt sind `-=`, `*=`, `/=` und `%=` sowie einige bitweise Operatoren, die im nächsten Absatz vorgestellt werden.

. . . . .

× Neben den arithmetischen Operatoren gibt es auch noch die so genannten *bitweisen Operatoren*. Mit ihnen können Binärzahlen verknüpft werden, dies geschieht stellenweise. Eine Stelle einer Binärzahl, die eine der Ziffern 0 oder 1 ist, heißt *Bit*, die Zahlen in ZETA haben 16 Bits. Man sagt, ein Bit ist gesetzt, wenn es den Wert 1 hat, und es ist gelöscht, wenn es null ist.

Das bitweise Und, `&`, verknüpft zwei Zahlen und setzt ein Bit des Ergebnisses, wenn dieses Bit in beiden Zahlen gesetzt ist. Das bitweise Oder, `|`, setzt ein Bit, wenn dieses Bit mindestens in einer der Zahlen gesetzt ist. Das bitweise Exklusiv-Oder, `^`, setzt ein Bit, wenn das Bit in nur einer der Zahlen gesetzt ist und das bitweise Nicht, `~`, manchmal auch *Einerkomplement* genannt, kehrt alle Bits in einer Zahl um. Zum Beispiel:

```
$$00100110 & $$10010111 == $$0000000000100110
$$00100110 | $$10010111 == $$0000000010110111
$$00100110 ^ $$10010111 == $$0000000010110001
           ~ $$10010111 == $$1111111101101000
```

Zwei weitere Operatoren, `<<` und `>>`, verschieben die Bits innerhalb einer Zahl. `a<<b` verschiebt die Bits in `a` um `b` Stellen nach links, `a>>b` nach rechts:

```
$$00100110 << 2 == $$10011000
$$00100110 >> 2 == $$00001001
```

Bits, die über den Rand hinausgeschoben werden, verfallen. Am anderen Rand werden immer nur Nullen eingefügt. Das höchste Bit, das *Vorzeichenbit*, bleibt dabei nicht erhalten. Eine Verschiebung um eine Stelle nach rechts entspricht einer Division durch zwei, eine Verschiebung um eine Stelle nach links einer Multiplikation mit zwei (wenn keine Bits „hinausgeschoben“ werden). Außerdem ist `a<<b` dasselbe wie `a>>-b`.

Als Abkürzung können die Operatoren `&=`, `|=`, `^=`, `<<=` und `>>=` benutzt werden.

1.5 Argumente von Routinen und Rückgabewerte

Die lokalen Variablen können nur innerhalb der Routine geändert werden. Es gibt aber eine Möglichkeit, ihnen Anfangswerte zu geben, die von null verschieden sind. Dazu muss man die lokalen Variablen in runden Klammern nach dem Bezeichner der Routine angeben, außerhalb der geschweiften Klammern, die die Anweisungen der Routine umschließen:

```
Routine Quadrat(x) {
    "Das Quadrat von ", num(x), " ist ",
    num(x*x), ".\n";
}
```

Dieser Routine kann nun beim Aufrufen ein Wert in runden Klammern übergeben werden: `Quadrat(5)` bedeutet: Rufe die Routine `Quadrat` auf und setze die erste lokale Variable auf den Wert 5. Bei diesem Aufruf wird „Das Quadrat von 5 ist 25.“ ausgegeben. Die Werte, die an die Routine übergeben werden, heißen *Argumente*.

Es können bis zu sieben Argumente an eine Funktion übergeben werden, die in der Klammer durch Kommas getrennt werden. Werden weniger Argumente übergeben, als die Routine Variablen hat, so sind die restlichen Variablen wie gehabt zu Beginn gleich null. Werden mehr Variablen übergeben, als in einer Routine definiert sind, verfallen die überzähligen Argumente.

Argumente können auch Variablen oder Ausdrücke sein, die folgenden Aufrufe sind alle erlaubt:

```
Quadrat(z);
Quadrat(2*a + 1);
Farbe(a, b, -1);
```

× Der letzte Aufruf ist legal, auch wenn in der Routine `Farbe` keine Argumente in runden Klammern, sondern drei lokale Variablen mit `lokal` definiert wurden. ZETA macht keinen Unterschied zwischen lokalen Variablen und Argumenten. Je Routine können nur fünfzehn Argumente und lokale Variablen definiert werden.

Auch wenn eine Variable übergeben wird, bleiben die Variablen in der aufgerufenen Routine immer lokal.

```
Routine doppelt(x) {
    x = x*2;
    num(x), "\n";
}

Hauptprogramm {
    lokal a;
    a = 2;
    doppelt(a);
    num(a);
}
```

Hier ist der Wert der Variable `a` nach Aufruf von `doppelt` immer noch 2, nicht etwa 4.

Routinen können nicht nur Werte empfangen, sie geben immer einen Wert an die aufrufende Routine zurück. Eine Routine `f` ist also mehr als nur ein Platzhalter für häufiger wiederkehrende Anweisungen. Sie empfängt einige Argumente, macht dann etwas, z.B. gibt sie Text aus oder führt Berechnungen durch, und schließlich übergibt sie einen Wert, den *Rückgabewert*, an die Routine `r`, aus der sie aufgerufen wurde, sie antwortet. Wie `f` den Wert erzeugt, interessiert `r` genauso wenig, wie `f` sich darum kümmert, was `r` mit dem zurückgegebenen Wert macht.

Wenn nichts anderes angegeben wird, ist dieser Wert null oder `falsch`. (Die beiden Systemkonstanten `falsch` und `wahr` haben die Werte 0 und 1.) Mit der Anweisung `antworte` wird die Routine beendet und der angegebene Wert wird als Antwort des Aufrufs zurückgegeben:

```
Routine doppelt(x) {
    antworte x*2;
}

Hauptprogramm {
    lokal a;
    a = 2;
    num(doppelt(a));
}
```

Mit der `num`-Anweisung wird nun der Rückgabewert der Routine `doppelt` ausgegeben, in diesem Fall 4. Der Rückgabewert der Routine wird dort eingesetzt, wo der Aufruf steht:

```
x = doppelt(3);              // x == 6;
x = 2*(doppelt(x+1)-1);      // x == 26;
```

Wenn der Funktionsaufruf alleine steht, wie im Beispiel unserer ersten Routine `Hallo`, wird der zurückgegebene Wert nicht benutzt, er wird quasi weggeworfen.

Eine besondere Form von `antworte` ist `Stop`, das `wahr` zurückgibt. `antworte` ohne nachfolgenden Wert gibt `falsch` zurück.

. . . . .

Eine besondere Art von Routinen sind *Systemroutinen* oder *Systemfunktionen*. Sie können wie Routinen aufgerufen werden, müssen aber nicht vom Programmierer definiert werden. Sie sind bereits in ZETA vorhanden.

Die Anweisung `Zufall(x)` ist eine Systemroutine von ZETA, die eine Zufallszahl von eins bis `x` erzeugt. `Zufall(6)` simuliert also einen Würfel.

Anders als bei benutzerdefinierten Routinen wird bei Systemroutinen überprüft, ob die Anzahl der Argumente bei der Übergabe stimmt und ob der Rückgabewert verwendet wird oder nicht. Die folgenden Anweisungen erzeugen einen Fehler beim Kompilieren:

```
Zufall(6);        // kein Rückgabewert
x = Zufall();     // kein Argument
```

Eine Liste aller Systemfunktionen befindet sich in Anhang A.

. . . . .

× Zufallszahlen werden trotz ihres Namens nicht wirklich zufällig, sondern mit einem Algorithmus, dem *Zufallszahlengenerator*, bestimmt. Die produzierte Zahlenfolge scheint zufällig, wiederholt sich aber, wenn auch erst nach langer Zeit. Die Zahlenfolge, die erzeugt wird, hängt von ihrem Ausgangswert, der so genannten *Saat*, ab. Wenn man `Zufall` mit einer negativen Zahl `-n` aufruft, so wird die Saat auf `n` gesetzt, der Rückgabewert ist null. Damit werden die Zufallszahlen vorhersehbar, was beim Testen von Spielen nützlich ist.

Wenn `Zufall` mehr als ein Argument hat, so wird aus den Argumenten eines zufällig ausgewählt. `Zufall(-1, -1, 3, 5)` gibt zu 50 Prozent –1 und zu je 25 Prozent 3 oder 5 zurück.

1.6 Bedingungen

Manche Anweisungen sollen nicht immer ausgeführt werden, sondern nur unter bestimmten Bedingungen. Um dies zu steuern, gibt es die `wenn`-Anweisung. Sie hat die Form

`wenn (`*Bedingung*`)` *Anweisung*

Die Anweisung wird nur ausgeführt, wenn die Bedingung wahr ist. Ein Beispiel dazu:

```
wenn (Punkte == 100) "Gewonnen!\n";
```

Wenn die Variable `Punkte` den Wert 100 hat, wird „Gewonnen!“ ausgegeben, und nur dann. Wenn `Punkte` einen anderen Wert hat, wird die Anweisung zum Ausgeben des Texts einfach übersprungen.

Der Operator `==` ist ein *Vergleichsoperator* und ist nicht mit dem Zuweisungsoperator `=` zu verwechseln. Code wie

```
wenn (Punkte = 100) "Gewonnen!\n";
```

ist zwar syntaktisch korrekt, macht aber etwas anderes: Der Variable `Punkte` wird der Wert 100 zugewiesen, und der Text wird ausgegeben, da ZETA diese Zuweisung als wahre Bedingung betrachtet. Dies kann das gewollte Verhalten sein, und deshalb erlaubt es ZETA auch, aber in diesem Fall ist es wahrscheinlicher, dass es einfach ein Tippfehler ist und ZETA gibt eine Warnung aus.

Der Operator `==` ist nur einer von sechs arithmetischen Operatoren:

`(a == b)`	ist gleich `b`
`(a != b)`	ist nicht gleich `b`
`(a > b)`	ist größer als `b`
`(a < b)`	ist kleiner als `b`
`(a >= b)`	ist größer als oder gleich `b`
`(a <= b)`	ist kleiner als oder gleich `b`

Außerdem gilt nur die Angabe einer Variable oder Konstante als Bedingung: `(Flagge)` ist falsch, wenn `Flagge` gleich null ist und ansonsten wahr.

Man kann mehrere Bedingungen zu einer verknüpfen, indem man die Operatoren `und`, `oder` und `nicht` benutzt:

```
wenn (a==0 oder b==0) "Golch";
wenn (nicht(x > 6 oder x <= 0)) "Flubis";
```

„Golch“ wird ausgegeben, wenn eine der Variablen `a` oder `b` gleich null ist, „Flubis“, wenn `x` zwischen 1 und 6 liegt.

Bedingungen, die mit `nicht` verneint werden, müssen wieder in runden Klammern stehen. `und` bindet stärker als `oder`, aber wie bei den Rechenoperationen können Klammern gesetzt werden, um die Bindung anzugeben. Anstelle von `und`, `oder` und `nicht` kann man auch die in C üblichen Operatoren `&&`, `||` und `!` verwenden.

Diese Verknüpfungen werten unter Umständen nur den linken Teil einer Bedingung aus. Wenn im Beispiel oben `a` gleich null ist, ist die Bedingung `(a==0 oder b==0)` bereits erfüllt, und `b==0` wird nicht mehr überprüft. Dies hat eine Bedeutung, wenn im rechten Teil eine Routine steht:

```
wenn (a!=0 und sinnvoll(a, 1)) ...
```

Wenn `a` null ist, wird `sinnvoll` nicht aufgerufen. Dies ist wichtig, wenn dort Text ausgegeben wird oder globale Variablen geändert werden.

Anstelle einer einzelnen Anweisung kann man nach der Bedingung auch mehrere Anweisungen, die in geschweifte Klammern gefasst sind, angeben. Man nennt dies einen *Anweisungs-* oder *Codeblock*:

```
wenn (a < 0 und b < 0) {
    a = 0;
    b = 0;
    "Warnung: a und b negativ.";
}
```

Die Einrückung des Blocks ist Geschmackssache, macht aber den Code leserlicher. Üblicherweise wird die öffnende Klammer auf die Zeile geschrieben, in der die `wenn`-Anweisung steht. Die nachfolgenden Anweisungen werden eingerückt und die abschließende Klammer bekommt eine eigene Zeile und steht direkt unter `wenn`. Diese Konvention wird in diesem Handbuch verwendet.

. . . . .

Eine nützliche Erweiterung von `wenn` ist `sonst`. Nach der Anweisung, die ausgeführt wird, wenn die Bedingung wahr ist, kann man nach `sonst` eine zweite angeben, die ausgeführt wird, wenn die Bedingung falsch ist:

`wenn (`*Bedingung*`)` *Anweisung*
`sonst` *Anweisung*

In jedem Fall wird eine, und nur eine, der beiden Bedingungen ausgeführt. Die Bedingung wird nur einmal überprüft:

```
wenn (a < 0) {
```

```
    a = 0;
    "a ist negativ";
} sonst "a ist null oder positiv";
```

Hier wird nur einer der Texte ausgegeben, auch wenn beim Erreichen von `sonst` `a` gleich null ist. Nach Ausführen des ersten Blocks wird der zweite übersprungen.

Vorsicht ist bei verschachtelten Bedingungen geboten:

```
wenn (a == 0)
    wenn (b == 0) "Klar.";
    sonst "Mehrdeutig.";
```

Auf welche Bedingung bezieht sich `sonst`? ZETA weist es dem letzten `wenn` zu. Um den Code übersichtlich zu machen, ist es aber besser, geschweifte Klammern zu setzen:

```
wenn (a == 0) {
    wenn (b == 0) "Klar.";
    sonst "Auch klar.";
}
```

Mit `wenn` kann man zwischen zwei Fällen unterscheiden, Vergleiche mit mehreren Werten sind eher umständlich. Eine Struktur, mit der man Variablen mit mehreren konstanten Werten vergleichen kann, ist `jenach`:

```
"Du hast ", Punkte, " Punkte, du bist also ein "
jenach (Punkte/50) {
    1: "Smutje";
    2: "Seeräuberanwärter";
    3: "Seeräuber";
    4: "Steuermann";
    5: "richtiger Piratenkapitän";
    sonst: "Deckschrubber";
}
```

Dies ist eine Abkürzung für

```
wenn (Punkte/50 == 1) "Smutje";
wenn (Punkte/50 == 2) "Seeräuberanwärter";
...
```

Vor dem Doppelpunkt dürfen jedoch nur konstante Werte stehen, auch Namen von Objekten und benutzerdefinierten Konstanten. Nach jedem Doppelpunkt kommen die Anweisungen, die für diesen Fall gelten. Sie stehen der Übersichtlichkeit halber nicht in geschweiften Klammern, auch nicht, wenn es mehrere sind. Die Fälle können in jeder Reihenfolge auftreten. Nur der Fall `sonst` muss der letzte sein, er muss aber nicht angegeben werden.

Bei den Konstanten vor dem Doppelpunkt können mehrere Angaben gemacht werden, die mit Kommas abgetrennt werden. Außerdem kann man Bereiche von Konstanten angeben, z.B. `1...40`. Dazu noch ein Beispiel:

```
"Du würfelst ", Augenzahl, ". ";
jenach (Augenzahl) {
    2...6:  "Die Bank gewinnt.";
            Geld = Geld − 5;
    11, 12: "Du gewinnst!";
            Geld = Geld + 10;
    sonst:  "Nichts passiert.";
}
```

1.7 Schleifen

Mit Hilfe von Schleifen kann man einen Anweisungsblock mehrmals ausführen. Die einfachste Schleifenanweisung ist `solange`:

`solange (`*Bedingung*`)` *Anweisung*

Hier wird die Bedingung geprüft. Ist sie wahr, so wird die Anweisung ausgeführt. Dies wird solange wiederholt, bis die Bedingung falsch ist. Ist die Bedingung bereits beim ersten Mal falsch, so wird der Block nie ausgeführt.

```
a = 10; x = 0;
solange (a > 0) {
    x = x + a;
    a−−;
}
num(x);
```

In diesem Beispiel werden alle Zahlen von 1 bis 10 summiert, wobei allerdings bei 10 angefangen wird zu zählen.

Eine weitere Schleife prüft die Bedingung erst nach dem Durchlaufen des Anweisungsblocks:

`wiederhole` *Anweisung* `bis (`*Bedingung*`)`

Hier wird der Block mindestens einmal durchlaufen, auch wenn beim ersten Mal die Bedingung wahr ist.

Bei Schleifen muss man aufpassen, dass die Bedingung – man spricht hier von einer *Abbruchbedingung* – irgendwann einmal erfüllt ist. Würde z.B. die Anweisung `a--` im

Beispiel oben fehlen, wäre a immer größer als null und Schleife wiederholte sich ununterbrochen. Dies darf nicht passieren, und man spricht hier von einer *Endlosschleife*.

Zwei elementare Endlosschleifen sind:

```
solange (wahr) ...
wiederhole ... bis (falsch)
```

. . . . .

Die Anweisung `Schleife` ist eine Abkürzung für `solange`-Schleifen, die besonders oft benötigt werden.

```
Initialisierung
solange (Bedingung) {
    ...
    Aktualisierung
}
```

kann abgekürzt werden zu

```
Schleife (Initialisierung; Bedingung; Aktualisierung)
```

Das Beispiel oben könnte also umgeschrieben werden zu:

```
x = 0;
schleife (a=10; a>0; a--) x += a;
num(x);
```

Innerhalb der runden Klammer müssen immer zwei Strichpunkte stehen, auch wenn einzelne Angaben fehlen können.

```
schleife (; a>0; a--) ...
```

Hier wird a nicht initialisiert, der momentane Wert von a wird benutzt.

```
schleife (a=0; ; a++) ...
```

Dies ist eine Endlosschleife.

```
schleife (a=0; a<n;) ...
```

Hier wird nicht aktualisiert. a muss seinen Wert also auf eine andere Art innerhalb des Anweisungsblocks ändern.

Neben den Abbruchbedingungen gibt es zwei weitere Möglichkeiten, aus einer Schleife herauszukommen. Mit ihnen kann man sogar aus einer Endlosschleife ausbrechen.

Die Anweisung `abbrechen` verlässt die Schleife sofort. Die folgende Schleife

```
schleife (a=0; a<5; a++) {
    wenn (a==3) abbrechen;
    num(a), " ";
}
```

schreibt lediglich „0 1 2“ auf den Bildschirm, da die Schleife abgebrochen wird, bevor „3“ geschrieben werden soll. Die Anweisung `weiter` hingegen bricht nur den jetzigen Durchgang ab:

```
schleife (a=0; a<5; a++) {
    wenn (a==3) weiter;
    num(a), " ";
}
```

Hier wird „0 1 2 4“ ausgegeben. Lediglich der Durchgang, in dem `a` den Wert 3 hat, wird übersprungen.

× Zusätzlich zu diesen Anweisungen kann man Schleifen mit Hilfe benutzerdefinierter Sprungmarken (oder *Labels*) und der Anweisung `springe` steuern. Eine Sprungmarke ist einfach ein Bezeichner, der hinter einem Punkt angegeben wird. Sie erzeugt keinen Code, sondern markiert nur eine Stelle im Programm. Der Bezeichner kann dann mit `springe` benutzt werden: Diese Anweisung springt einfach an den Punkt, der durch die passende Sprungmarke gekennzeichnet ist.

Als primitives Beispiel wird hier eine Endlosschleife erzeugt:

```
i = 0;
.dahin;
num(++i), ". Durchgang\n";
springe dahin;
```

Intern werden Schleifen und Bedingungen in ZETA mit Sprungmarken verwirklicht, die dem Programmierer nicht zugänglich sind. Im Allgemeinen ist es besser, die regulären Anweisungen für Schleifen zu benutzen, da viele Sprungmarken den Quelltext unübersichtlich machen.

In manchen Fällen können Sprungmarken aber nützlich sein. Zum Beispiel, um größere Codeblöcke zu überspringen oder um `abbrechen` und `weiter` für verschachtelte Schleifen zu implementieren:

```
schleife (i=1; i<=imax; i++) {
    schleife (j=1; j<=jmax; j++) {
        wenn (i*j == 8) springe ganzWeiter;
        num(i), " ", num(j), " " , num(i*j), "\n";
        wenn (i*j == 25) springe ganzAbbrechen;
    }
    .ganzWeiter;
}
.ganzAbbrechen;
```

Sprungmarken sind immer lokal, d.h. man kann nicht aus einer Routine heraus springen. (Obwohl die Spezifikation der *Z*-Maschine dies erlaubt.)

× Mit der Anweisung `beenden` kann man den Interpreter verlassen, so als ob das Ende des Hauptprogramms erreicht wäre. Diese Anweisung sollte allerdings nur als letzter Ausweg in verfahrenen Situationen benutzt werden.

1.8 Felder

Man kann Variablen in Strukturen zusammenfassen. Die einfachste Methode, dies zu tun, sind Felder. Dies sind Listen von Variablen, deren Einträge mit ihrer laufenden Nummer angesprochen werden können.

Zum Beispiel können wir so einen Stoß Karten erzeugen:

```
Feld Karten[32];
```

Dieses Feld hat 32 Einträge, die mit

```
Karten[0], Karten[1], Karten[2], ... , Karten[31];
```

angesprochen werden können. Zu Beginn haben diese Einträge alle den Wert null. Nun soll aber der Index die Position im Stoß angeben, der jeweilige Eintrag den Wert der Karte, der ebenfalls zwischen 0 (Herz Sieben) und 31 (Kreuz As) liegt:

```
Routine KartenOrdnen {
    lokal i;
    schleife (i=0; i<32; i++) Feld[i] = i;
}
```

Nun sind die Karten alle schön geordnet, jede Karte taucht genau einmal im Stoß auf. Nun wollen wir sie mischen:

```
Routine TauscheKarten() {
    lokal a, b, c;
    solange (a == b) {
        a = Zufall(32) - 1;
        b = Zufall(32) - 1;
    }
    c = Karten[a];
    Karten[a] = Karten[b];
    Karten[b] = c;
}

Routine KartenMischen(n) {
    lokal i;
    schleife (i=0; i<n; i++) TauscheKarten();
}
```

Mit `TauscheKarten(a, b)` kann man zwei Karten, die zufällig bestimmt werden, im Stapel austauschen. Mit `KartenMischen(n)` wird dies `n` Mal gemacht. Nach `KartenMischen(100)` sollte der Stoß gut gemischt sein.

Ein wenig Kosmetik noch zur Ausgabe des gemischten Stoßes:

```
Routine Kartenname(n) {
    jenach (n/8) {
        0: "Herz ";
        1: "Karo ";
        2: "Pik ";
        3: "Kreuz ";
    }
    jenach (n%8) {
        0...3: num(n%8 + 7);
        4: "Bube";
        5: "Dame";
        6: "König";
        7: "As";
    }
}

Hauptprogramm {
    lokal i;
    KartenOrdnen();
    KartenMischen(100);

    "Die Karten sind nun in folgender Reihenfolge:\n";
    schleife (i=0; i<32; i++)
        Kartenname(Karten[i]), "\n";
}
```

!! Etwas, worauf man bei Feldern achten muss, ist, dass man nicht versucht, ausserhalb der Feldgrenzen zu lesen oder noch schlimmer, zu schreiben. Wer einmal in `Karten[32]` oder `Karten[-1]` nachschaut, findet dort wahrscheinlich Werte, mit denen er nichts anfangen kann. Wer dorthin schreibt, zerstört Daten anderer Felder, im schlimmsten Fall sogar die Spieldatei. Ob ein Feldindex im erlaubten Bereich liegt, wird von ZETA nicht überprüft. Es ist am Programmierer, dies sicherzustellen.

. . . . .

Es gibt eine zweite Möglichkeit, Felder zu definieren. Anstatt die Dimension anzugeben, kann man eine Liste von Werten nach einem Gleichzeichen angeben, mit denen die Feldeinträge vorbelegt werden. Die Dimension des Feldes ist dann gleich der Anzahl der Einträge:

```
Feld zweiHoch = $01, $02, $04, $08, $10, $20, $40, $80;
```

Dieses Feld hat die acht Einträge `zweiHoch[0]` bis `zweiHoch[7]`, die die Werte der dazugehörigen Zweierpotenzen haben.

Eine letzte Möglichkeit ist, eine Zeichenkette in Anführungszeichen anzugeben. Dann ist die Dimension des Feldes gleich der Anzahl der Zeichen plus eins, die Werte der Einträge entsprechen dem Wert der Zeichen im ZSCII-Code, denen eine Null zum Markieren des Endes angehängt wird:

```
Feld geheimesWort = "uruguay";
```

Diese Deklaration ist gleichwertig mit

```
Feld geheimesWort =
    'u', 'r', 'u', 'g', 'u', 'a', 'y', 0;
```

Die Einträge des Feldes können sich – genau wie bei anderen Feldern, aber im Gegensatz zu anderen Texten in Anführungszeichen – ändern

× Folgende Definition macht hingegen etwas anderes: Sie erzeugt ein Feld mit den Adressen von fünf konstanten Zeichenketten:

```
Feld LänderMitU = "Uganda", "Ukraine", "Ungarn",
    "Uruguay", "Usbekistan";
```

Die zweite, komplexere Datenstruktur, Objekte, werden im nächsten Kapitel behandelt.

× Ein Feld kann mit der Direktive `String` anstelle von `Feld` definiert werden. Dann sind die Einträge nur ein Byte lang und können nur Werte von 0 bis 255 annehmen. Diese Art von Feldern wird oft für Zeichenketten verwendet, da die Codes für darstellbare Zeichen in

diesem Bereich liegen. Das „Uruguay“-Beispiel wäre typischerweise als `String` deklariert worden, aber auch das Karten-Beispiel hätte mit dieser Speicherplatz sparenden Methode funktioniert. Die Einträge von Feldern, die mit `Feld` deklariert werden, sind dagegen normale ZETA-Zahlen, die zwei Bytes belegen.

× Neben der Möglichkeit, zu jedem Feld einen Index in eckigen Klammern anzugeben, gibt es zwei weitere Operatoren, um auf die Einträge in einem Feld zuzugreifen. Mit `->` liest man von einem Feld, als ob es ein `String` wäre, mit `-->` wie von einem `Feld`, das normale ZETA-Zahlen speichern kann.

!! Dies birgt besonders für Einsteiger einige Gefahren. Normal weiß ZETA, ob ein Feld ein String ist oder nicht und benutzt immer die richtige Methode, um die Einträge zu lesen oder sie zu ändern. Wenn der Feldbezeichner aber eine Variable ist, kann ZETA nicht wissen, um was für ein Feld es sich handelt. Es liest dann die Einträge, als ob es *kein* String wäre, und dann bekommt man bei Strings unsinnige Einträge geliefert. Dann muss auf diese Operatoren zurückgegriffen werden. Man kann so zum Beispiel eine Routine schreiben, die einen beliebigen String, der in einem mit `String` deklarierten Feld liegt, ausgibt:

```
Routine sText(textfeld) {
    lokal i;
    solange (textfeld->i)
        char(textfeld->(i++));
}
```

Für solche Aufgaben stehen aber bereits fertige Routinen zur Verfügung, die leicht eingebunden werden können. Einsteiger brauchen sich also keine Gedanken zu machen und sollten immer den sicheren Weg der Indizierung mit eckigen Klammern wählen.

. . . . .

× Die Systemfunktion `kopiereFeld(a, b, n)` kopiert `n` Bytes von Feld `a` nach Feld `b`. Auf diese Weise kann man schnell Datenblöcke übertragen, ohne eine Schleife benutzen zu müssen:

```
String eins "Regenwald";
String zwei "Nebelhorn";

...
kopiereFeld(zwei, eins, 5);
```

Der String `eins` lautet nun „Nebelwald“.

Mit der Funktion `löscheFeld(a, n)` setzt man `n` Bytes im Feld `a` auf den Wert null.

1.9 Texte

Die Ausgabe von Text ist bis jetzt schon oft vorgekommen, ohne dass sie erklärt wurde. Eine Anweisung, die nur aus einem Text in Anführungszeichen besteht, gibt diesen Text aus:

```
"Einstmals, vor vielen, vielen Jahren lebte im
Morgenland ein mächtiger König. Er herrschte über große
Länder und wohnte in einem Schloss aus weißem Marmor, der
in der Sonne blitzte wie getriebenes Silber.";
```

Der Text, der in Anführungszeichen eingeschlossen ist, kann dabei über mehrere Zeilen gehen. Der Zeilenumbruch sowie alle Leerstellen, die vor oder hinter dem eigentlichen Text stehen, werden dabei nur zu einem Leerzeichen umgewandelt, so dass die Anweisung oben schreibt:

> „Einstmals, vor vielen, vielen Jahren lebte im Morgenland ein mächtiger König. Er herrschte über große Länder und wohnte in einem Schloss aus weißem Marmor, der in der Sonne blitzte wie getriebenes Silber.“

Zeilenumbrüche in Texten werden mit `\n` geschrieben. Absätze werden durch eine Leerzeile im Text oder durch `\p` gekennzeichnet:

```
"Gruselett

Der Flügelflagel gaustert\n
durchs Wiruwaruwolz,\n
die rote Fingur plaustert\n
und grausig gutzt der Golz.

-- Christian Morgenstern";
```

Hier wird nach jeder Zeile des Gedichts umgebrochen. Da das letzte Zeichen auf der Zeile ein `\n` ist, wird hier kein Leerzeichen eingefügt. Ein Absatz wird, wenn nichts anderes angegeben wird, durch eine Leerzeile vom nächsten getrennt und hat keinen Einzug.

Wenn man Anführungsstriche schreiben will, so muss man sie innerhalb des Textes doppelt setzten, damit ZETA weiß, dass es nicht das Ende des Textes sein soll:

```
"""Das"", sagt Holmes, ""ist elementar, mein
lieber Watson.""";
```

Weiterhin gibt es einige besondere Zeichen, die wie die Notationen für den Zeilenumbruch und für einen Absatz mit dem *Backslash* \ beginnen. Diese sind hauptsächlich dazu da, um Zeichen mit Umlauten oder Akzenten auf Computern zu schreiben, auf deren Tastatur diese

Buchstaben nicht vorhanden sind.

Zunächst gibt es Akzente, die auf den nachfolgenden Buchstaben gesetzt werden. Dabei sind nicht alle Kombinationen erlaubt, anstatt der angegebenen können auch die entsprechenden Großbuchstaben angegeben werden.

`\:`	Umlaut oder Trema auf a, e, i, o, u und y
`\'`	*Accent aigu* auf a, e, i, o, u und y
`\‘`	*Accent grave* auf a, e, i, o und u
`\^`	Zirkumflex auf a, e, i, o und u
`\~`	Tilde auf a, n und o

Schließlich gibt es noch besondere Zeichenkombinationen, die spezielle Zeichen oder Ligaturen erzeugen:

`\ss`	scharfes ß
`\aa \AA`	schwedisches å
`\/o \/O`	skandinavisches ø
`\ae \AE`	skandinavische *Æ*-Ligatur
`\oe \OE`	französische Œ-Ligatur
`\cc \CC`	Cedille
`\et \ET`	isländisches Eth
`\th \TH`	isländisches Thorn
`\>> \<<`	spitze Anführungszeichen
`\!! \??`	spanische Ausrufe- und Fragezeichen
`\LL`	Pfund Sterling

Statt dieser Kombinationen können immer die passenden Buchstaben direkt angegeben werden. Wer also eine deutsche Tastatur hat, kann beruhigt `ä` anstelle von `\:a` schreiben. Auch die meisten Zeichen mit Akzenten lassen sich mit der deutschen Tastatur erzeugen.

Es gibt auch einige Zeichen, die sich nur darstellen lassen, wenn man sie mit einem vorangestellten *Backslash* „schützt“, da sie eine Bedeutung innerhalb von Texten haben:

`\\`	*Backslash*
`\"`	Anführungszeichen (Alternative zu `""`)
`\{`	geschweifte Klammer auf
`\}`	geschweifte Klammer zu
`\ `	(*Backslash* und Leerzeichen) geschütztes Leerzeichen

Das geschützte Leerzeichen verhindert, dass vor und nach einem Zeilenumbruch im Quelltext alle Leerzeichen bis zum ersten Zeichen, das kein Leerzeichen ist, ignoriert werden. Wenn man Tabellen oder einfache Skizzen drucken möchte, ist das oft unerwünscht, und man kann

den Anfang oder das Ende einer Zeile mit „\ “ markieren.

. . . . .

× So genannte *Kurzstrings* erlauben es, veränderliche Zeichenfolgen innerhalb von Texten auszugeben. Man kann mit der Direktive

```
Kurzstring Bezeichner "Text"
```

Strings definieren, die als Platzhalter fungieren. Sie können später zwischen geschweiften Klammern in Texten angegeben werden. Der definierte String wird dann dort bei der Ausgabe des Texts eingesetzt.

Das ist nützlich, um häufig auftauchende Strings abzukürzen oder sie flexibel zu halten. Wenn z.B. der Spieler einen Begleiter hat, der jenachdem, ob er selbst ein Mann oder eine Frau ist, Iwan oder Natascha heißt. Dann kann man folgendes definieren:

```
Kurzstring Iwan "Iwan";
```

Nun kann man die Abkürzung `{Iwan}` in Texten verwenden:

```
"{Iwan} folgt dir auf dem Fuße.";
```

Der Wert des Strings kann dann während des Spiels geändert werden mit

```
Kurzstring Iwan "Natascha";
```

ZETA verwendet einige dieser Strings, um Beugungen in Objektnamen einfach angeben zu können.

Da die Bezeichner für Kurzstrings immer nur in einem eindeutigen Kontext verwendet werden, können sie Bezeichner haben, die bereits vergeben wurden. So kann es z.B. einen Kurzstring mit dem Bezeichner `Westen` geben, auch wenn es bereits eine Routine oder Variable mit diesem Namen gibt.

. . . . .

Text zwischen Anführungsstrichen ist nur eine Art, etwas auszugeben. Zur Ausgabe von Text gibt es einige Systemfunktionen. Eine davon wurde bereits vorgestellt:

```
num(10*Punkte);
```

Diese Systemfunktion gibt den numerischen Wert von `10*Punkte` als Dezimalzahl aus. Weitere Funktionen geben Werte in bestimmten Formaten aus:

`Text(x)`	gibt den Text an Adresse `x` aus
`Vokabel(x)`	gibt die Vokabel an Adresse `x` aus
`Char(x)`	gibt das Zeichen mit dem ZSCII-Code `x` aus

Natürlich kann der Programmierer eigene Routinen zur Textausgabe schreiben, zum Beispiel die Routine `Kartenname(x)` aus dem vorigen Abschnitt.

Die Ausdrücke, die man ausgeben möchte, kann man auch in einer Anweisung zusammenfassen, wenn man sie mit Kommas voneinander trennt.

```
"Du ziehst eine Karte: ", Kartenname(0), ".\p";

"Du ziehst eine Karte: "; Kartenname(0); ".\p";
```

Beide Zeilen geben denselben Text aus. Die erste Zeile ist jedoch eine Anweisung mit drei Ausdrücken, die zweite Zeile besteht aus drei einzelnen Anweisungen. Dies kommt bei einer weiteren Anweisung zum Tragen: `Stop`, das aus der Routine zurückspringt und `wahr` zurückgibt, kann auch mit einem Text versehen werden. Dann gibt es diesen Text aus, bevor es wahr zurückgibt. Dieser Text kann auch zusammengesetzt sein, wenn man dazu Kommas benutzt. Eine Zeile wie

```
Stop "Du kommst nicht an "; den(aObj); " heran.";
```

erzeugt allerdings die Fehlermeldung „Diese Anweisung wird nie ausgeführt“ bei der zweiten Anweisung, da nach `Stop` die Routine bereits verlassen wurde.

. . . . .

Eine Konstante, die einen Text enthält, kann wie ein Text in Anführungszeichen verwendet werden:

```
Konstante Absatz = "\p    ";

Routine meinFormat {
    "Es folgt ein eingerückter Absatz", Absatz,
    "Dies ist ein eingerückter Absatz.";
}
```

. . . . .

ZETA erlaubt es, Unicode-Zeichen zu verwenden, obwohl nur moderne Interpreter sie anzeigen können. Dazu gibt es zwei Möglichkeiten, einmal können hexadezimale Unicodes in geschweiften Klammern angegeben werden:

```
"""Das macht 12,50 {$20ac}, bitte."""
```

Dabei muss das Dollarzeichen angegeben werden, um diese Notation von den Kurzstrings zu unterscheiden. Wenn der Interpreter das Eurozeichen mit dem Unicode `$20ac` nicht korrekt darstellen kann, wird hier ein Fragezeichen ausgegeben, was unschön ist. Außerdem ist die Anzahl der so ausgegebenen Zeichen beschränkt. Der ZSCII-Zeichensatz hat Raum für 97 so genannte „erweiterte Zeichen“. Normalerweise sind dies die lateinischen Buchstaben mit Akzenten sowie einige Sonderzeichen, etwa das Eszett, das Pfund-Zeichen oder spitze Anführungszeichen. Diese Zeichen sind vorbelegt und nehmen 69 Plätze weg. Die restlichen 28 Zeichen werden automatisch aufgefüllt, wenn sie benötigt werden. Wenn der Platz aufgebraucht ist, gibt es eine Fehlermeldung. (Der erweiterte Zeichensatz kann optimiert werden, mehr dazu in Abschnitt 4.3).

Außerdem gibt es die Systemfunktion `Unicode`, mit der man ein beliebiges Unicode-Zeichen ausgeben kann:

```
"""Das macht 12,50 ", Unicode($20ac, "Euro"), "."""
```

Der Text wird geschrieben, wenn das Unicode-Zeichen nicht ausgeben werden kann, weil der Interpreter nicht dem *Z*-Code-Standard 1.0 entspricht oder weil dieses Zeichen nicht implementiert ist, zum Beispiel bei einem Konsoleninterpreter. Diese zweite Angabe kann auch ein Zeichen in Hochkommas sein, etwa `'E'`. Fehlt diese Angabe, wird nichts geschrieben, wenn der Unicode nicht verfügbar ist. Die Benutzung dieser Funktion ist nicht beschränkt, es können beliebig viele verschiedene Zeichen mit `Unicode` ausgegeben werden. Zusammen mit der Kontrolle darüber, ob das Zeichen existiert oder nicht, ist dies ein Vorteil für längere Texte, die in Russisch oder Griechisch (oder in Tolkien’schem Elbisch) ausgegeben werden sollen.

Zeichen, die mit `Unicode` ausgegeben werden sollen, sind nicht im ZSCII-Zeichensatz repräsentiert und können nicht gelesen werden. Wenn die erste Methode der Ausgabe verwendet wird, ist das entsprechende Zeichen auch für die Eingabe definiert, auch wenn exotische Zeichen sich nicht eifach über die Tastatur eingeben lassen.

× Das Eurozeichen ist aber auf den meisten Tastaturen vorhanden. Der von Windows verwendete ASCII-Code 128 entspricht übrigens nicht dem Standard ISO 8859-1, so dass hier auf den (korrekten) Unicode `$20ac` zurückgegriffen werden sollte.

Die Systemfunktion `Unicode` gibt wahr oder falsch zurück, je nachdem, ob das Zeichen existiert. Wenn das zweite Argument eins ist, wird in keinem Fall ein Zeichen ausgegeben.

man kann also die Verfügbarkeit von Unicode-Zeichen testen:

```
wenn (Unicode($20ac), 1) {
    Kurzstring Euro "{$20ac}";
} sonst {
    Kurzstring Euro "Euro";
}
```

Wenn diese Sequenz in einer Initialisierungroutine angegeben wird, kann innerhalb von Strings mit der Notation `{Euro}` je nach Interpreter „€“ oder „Euro“ ausgegeben werden. Wenn der Unicode existiert, ist das Eurozeichen auch für die Eingabe definiert.

× Wenn man ein Unicode-Zeichen als einzigen Eintrag in Hochkommas benutzt, gibt man damit eine ZSCII-Konstante an. Genau wie `'\:a'` der ZSCII-Wert des Buchstabens „ä“ ist, ist `'{$a9}'` der ZSCII-Wert des Copyright-Zeichens. Dieser Wert ist allerdings verschieden von `$a9` oder 169, er liegt im Bereich des erweiterten ZSCII-Zeichensatzes von 155 bis 251.

Obwohl es vermutlich wenig nützlich ist, ist es möglich, diese erweiterten Zeichen in Vokabeln zu benutzen. Die Konstante `'{$a7}\'` ist die Vokabel, die nur aus dem Paragraphenzeichen „§“ besteht.

2 Objekte und Klassen

2.1 Das Konzept der Objekte

Objekte sind Datenstrukturen, in denen Variablen und Routinen zusammengefasst sind. Dies ist gerade in Adventures praktisch: Jeder Gegenstand und jeder Raum wird in ZETA durch ein Objekt dargestellt.

Im Großen und Ganzen sieht die Definition eines Objekts so aus:

```
Objekt Name { ... }
```

(Die genaue Definition eines Objekts wird später beschrieben.) Alles, was zwischen den geschweiften Klammern steht und hier mit drei Punkten abgekürzt wird, sind die Daten, die zu diesem Objekt gehören.

Die Konstante `nichts`, die den Wert null hat, bedeutet „kein Objekt“. Und tatsächlich ist `nichts` kein Objekt: Es hat keine Eigenschaften, und es ist nicht erlaubt, Daten von `nichts` zu lesen oder gar zu ändern.

2.2 Der Objektbaum

Genau wie die Gegenstände im Spiel eine bestimmte Lage in Bezug auf andere Objekte haben, sind auch die Objekte in ZETA einander zugeordnet. Diese Zuordung heißt *Objektbaum*.

Dieser Objektbaum ähnelt einem Stammbaum: Jedes Objekt hat eine Mutter, eine Tochter und eine Schwester, die entweder ein anderes Objekt oder `nichts` sind. Tochter heißt hier immer „älteste Tochter“, Schwester bedeutet „nächstältere Schwester“. Ein typischer Objektbaum sähe etwa so aus:

```
Zimmer
  |
Tisch    -> Spieler  -> Stuhl
  |           |
Zeitung     Apfel    -> Rucksack -> Schlüssel
                            |
                         Luftpumpe
```

`Tisch`, `Stuhl` und `Spieler` sind Töchter ihrer Mutter `Zimmer`. Allerdings ist nur der `Tisch` *die* Tochter von `Zimmer`. Der `Schlüssel` ist die Schwester des `Rucksack`, der wiederum die Schwester vom `Apfel` ist.

Die drei Routinen `Mutter(x)`, `Tochter(x)` und `Schwester(x)` bestimmen die Verwandten eines Objekts `x`. Die Funktion `Kinder(x)` gibt an, wieviele Kinder `x` hat. In unserem Beispiel wären:

```
Mutter(Tisch) == Zimmer
Mutter(Rucksack) == Spieler
Tochter(Apfel) == nichts
Tochter(Tisch) == Zeitung
Schwester(Spieler) == Stuhl
Kinder(Spieler) == 3
```

Diese Funktionen können nur auf Objekte angewandt werden, auf nichts anderes, insbesondere nicht auf die Konstante `nichts`.

. . . . .

Es gibt zwei Möglichkeiten, ein Objekt zu Beginn im Objektbaum zu platzieren. Die erste ist, die Position eines Objekts relativ zu den vorher definierten anzugeben:

```
Objekt Zimmer { ... }
Objekt -> Tisch { ... }
Objekt -> -> Zeitung { ... }
Objekt -> Stuhl { ... }
```

Diese Definition entspricht dem Beispiel oben, jedoch ohne den `Spieler`. Ein Objekt ohne Pfeile hat keine Mutter, ein Objekt mit einem Pfeil hat das zuletzt ohne Pfeile definierte Objekt als Mutter, ein Objekt mit zwei Pfeilen das zuletzt mit einem Pfeil definierte, und so weiter.

Allgemein ausgedrückt: Die Anzahl der Pfeile gibt die Verschachtelungstiefe der Objekte an, ist sie größer als null, so ist das zuletzt mit einem Pfeil weniger definierte Objekt die Mutter. Alle hintereinander mit gleichviel Pfeilen definierten Objekte sind Schwestern.

Die Alternative hierzu ist, die Mutter mit `in` anzugeben:

```
Objekt Zimmer { ... }
Objekt Tisch in Zimmer { ... }
Objekt Stuhl in Zimmer { ... }
Objekt Zeitung in Tisch { ... }
```

Bei dieser Variante ist die Reihenfolge der Definitionen egal. Man kann beide Varianten gemischt einsetzen, aber es ist besser, wenn man sich für eine der beiden Arten der Objektdefinition entscheidet. Bei der zweiten Variante ist zu beachten, dass die Mutter nur hinter `in` angegeben werden kann. (Man ist immer versucht, etwas wie `auf Tisch` oder `im Zimmer` zu schreiben.)

. . . . .

Natürlich kann die Position eines Objekts innerhalb des Baums nachträglich geändert werden, etwa, wenn der Spieler sich in einem Adventure in einen anderen Raum begibt oder Dinge aufhebt und hinlegt. Dazu gibt es zwei Anweisungen:

`verschiebe` *Objekt* `nach` *Objekt*

verschiebt das erste Objekt im Baum, so dass es nun die Tochter des zweiten ist. Dabei werden alle Kinder des ersten Objekts mit verschoben. Nach Verschieben des Spielers auf den Stuhl sähe unser Baum so aus:

```
Zimmer
  |
Tisch    -> Stuhl
  |          |
Zeitung     Spieler
              |
            Apfel    -> Rucksack -> Schlüssel
                          |
                        Luftpumpe
```

Beim Verschieben darf keins der Objekte `nichts` sein. Um ein Objekt aus dem Baum herauszunehmen, muss man daher die Anweisung

`entferne` *Objekt*

verwenden. Hier wird das Objekt mit seinen Kindern so verschoben, dass es keine Mutter mehr hat. Wieder ein Beispiel, der Baum nach Entfernen des Rucksacks sieht so aus:

```
Zimmer                                Rucksack
  |                                     |
Tisch    -> Stuhl                     Luftpumpe
  |          |
Zeitung     Spieler
              |
            Apfel    -> Schlüssel
```

Man erhält zwei voneinander unabhängige Bäume. In einem typischen Spiel gibt es viele

Bäume, deshalb ist es nicht ganz korrekt, von *dem* Objektbaum zu sprechen.

!! Mit `verschiebe` kann man den Baum in „unmögliche“ Zustände versetzen, wenn man versucht, ein Objekt in ein zweites zu verschieben, das eine Tochter (oder Enkelin oder Urenkelin) des ersten ist, etwa mit `verschiebe Tisch nach Zeitung`. Solche Operationen sind zwar möglich, hinterlassen aber einen beschädigten Objektbaum und müssen daher unbedingt vom Programmierer vermieden werden.

. . . . .

Schließlich gibt es noch eine Bedingung, mit der man prüfen kann, ob ein Objekt eine Tochter eines anderen ist:

`(`*Objekt* `in` *Objekt*`)`

Diese Bedingung ist wahr, wenn das erste Objekt ein direktes Kind des zweiten ist. `(x in y)` ist also nur eine übersichtlichere Schreibweise für `(Mutter(x)==y)`. Die Verneinung dazu ist `nichtin`. Das erste Objekt, darf nicht `nichts` sein, das zweite schon:

```
(Rucksack in nichts)
```

prüft, ob der `Rucksack` vom Baum abgespalten wurde und die Wurzel eines eigenen Baums ist.

. . . . .

Es gibt eine Schleife, die alle Objekte nacheinander durchläuft:

`alleObjekte` *Variable* `{` *Anweisungen* `}`

(Anstelle eines Blocks von Anweisungen in geschweiften Klammern kann wie üblich auch eine einzelne Anweisung ohne Klammern stehen.)

Diese Schleife zählt die Variable einfach durch alle Objekte mit ihren internen Nummern durch. Dies kann in einem großen Spiel lange dauern. Deshalb ist es für einen Fall

```
alleObjekte x wenn (x in Eimer) {
    ...
}
```

besser, nur die Kinder des Objekts zu durchlaufen. Dazu benutzt man eine `solange`-Schleife:

```
x = Tochter(Eimer);
```

```
solange (x) {
    ...
    x = Schwester(x);
}
```

!! Während `x` die Schleife durchläuft, darf es nicht im Objektbaum bewegt werden, da sich dann das Schwesterobjekt ändert und die letzte Anweisung im Block nicht richtig funktioniert. Dann muss man doch `alleObjekte` oder benutzen oder das Schwesterobjekt vor dem Verschieben zwischenspeichern:

```
x = Tochter(Eimer);
solange (x) {
    y = Tochter(x);
    verschiebe x nach Kübel;
    x = y;
}
```

2.3 Attribute

Bis jetzt sind unsere Objekte noch gesichtslos: Sie haben einen Platz im Objektbaum, sonst nichts. Das ändert sich, wenn wir den Teil zwischen den geschweiften Klammern definieren.

Die einfachste Art von Daten, die ein Objekt haben kann, sind Attribute. Ein Attribut ist eine Eigenschaft, die ein Objekt hat oder nicht. Bevor man ein Attribut in einer Objekdefinition benutzen kann, muss man es deklarieren, zum Beispiel:

```
Attribut essbar;
```

Nun hat jedes definierte Objekt eine Information darüber, ob es `essbar` ist oder nicht. Attribute werden in der Objektdefinition mit `ist` vergeben:

```
Objekt Apfel {
    ist essbar;
}
```

Attribute, die nicht angegeben wurden, besitzt ein Objekt zu Beginn des Spiels auch nicht. Es können mehrere Attribute angegeben werden, die mit Kommas getrennt sind. Mit einem vorangestellten `nicht` oder `!` kann ein Attribut als nicht gesetzt definiert werden. Dies kann bei der Definition von Klassen nötig sein, um bereits vergebene Attribute wieder zu löschen, hat aber bei normalen Objekten keine Wirkung:

```
Objekt Truhe {
    ist verschließbar, zumöffnen, nicht offen;
```

```
}
```

Später kann man die Attribute eines Objekts mit `mache` ändern:

```
mache Stahltür offen;
mache Stahltür nicht offen;
```

machen die `Stahltür` auf und zu. Auch hier können wieder mehrere Attribute, verneint oder nicht, und durch Kommas getrennt, angegeben werden:

```
mache Apfel !essbar, verfault, vergiftet;
```

Um zu prüfen, ob ein Objekt ein Attribut hat, gibt es die Bedingung:

(*Objekt* `ist` *Attribut*)

So kann man mit `(Apfel ist essbar)` die Tauglichkeit des `Apfel`s zum Verzehr testen. Eine Alternative hierzu wäre die Verneinung `(Apfel istnicht essbar)`.

2.4 Eigenschaften von Objekten

Außer den Attributen besitzen Objekte so genannte Eigenschaften, die komplexer sind als das bloße Vorhandensein von Attributen.

Eine Eigenschaft ist eine Variable, die für alle Objekte, bei denen sie definiert ist, denselben Namen, aber nicht unbedingt denselben Wert hat. Folgende Definition

```
Objekt Apfel {
    ist essbar;
    Nährwert = 4;
}

Objekt Braten {
    ist essbar;
    Nährwert = 20;
}
```

erzeugt zwei Objekte, die einen unterschiedlichen `Nährwert` haben. Eine Eigenschaft wird mit ihrem Namen, eventuell gefolgt von einem Wert nach einem Gleichzeichen, definiert. Bevor sie benutzt werden kann, muss sie mit

```
Eigenschaft Nährwert ;
```

definiert werden. Man kann sich mit der Notation `Apfel.Nährwert` auf den Wert der Eigenschaft beziehen, der deutlich verschieden ist von `Braten.Nährwert`.

Nicht jedes Objekt hat jede Eigenschaft. Das Objekt

```
Objekt Mistgabel;
```

zum Beispiel kommt ohne Angabe des `Nährwert`s aus. Trotzdem kann die Eigenschaft `Mistgabel.Nährwert` gelesen werden, sie hat den Wert null. Mit

```
Eigenschaft Nährwert = -1;
```

könnten wir allen Objekten, die keinen `Nährwert` haben oder bei denen in der Definition des `Nährwerts` kein Wert mit = zugewiesen wurde, automatisch einen von –1 geben.

Beim Zuweisen von Eigenschaften, die ein Objekt nicht besitzt, gibt es jedoch Probleme:

```
Mistgabel.Nährwert = -5;
```

führt zu einem Laufzeitfehler. Deshalb kann man mit der folgenden Bedingung prüfen, ob ein Objekt eine bestimmte Eigenschaft besitzt:

(*Objekt* `hat` *Eigenschaft*)

. . . . .

Eigenschaften von Objekten können nicht nur Zahlen sein, sondern auch Routinen, Zeichenketten, andere Objekte oder Felder. Ein typisches Objekt sähe etwa so aus:

```
Objekt Möhre {
    Name = "Möhre";
    Vokabular = 'möhre', 'karotte', 'mohrrübe';
    Beschreibung = {
        wenn (Möhre ist verfault)
            "Ein unappetitliches, braunes Ding.";
        sonst "Die Karotte strahlt dich in
            knackigem Orange an.";
    }
    Nährwert = {
        wenn (Möhre ist verfault) antworte -1;
        antworte 6;
    }
    ist essbar, nicht verfault;
}
```

Die Eigenschaften `Besch` und `Nährwert` sind hier Routinen. Der `Name` ist eine Zeichenkette, und `Vokabular` ist ein Feld mit drei Einträgen.

Routinen, die Eigenschaften von Objekten sind – oft sagt man auch *Methoden* dazu – sind wie normale Routinen. Sie geben `falsch` zurück, wenn nichts anderes gesagt wird, und sie können Argumente empfangen.

Wenn man aus einer Eigenschaft mit mehreren Einträgen liest, so ist das Ergebnis immer der erste Eintrag. Solche Eigenschaften können aber auch wie Felder benutzt werden. Die Einträge haben Indizes, die bei null anfangen, und die man der Eigenschaft in eckigen Klammern nachstellen kann. Mit `.#` bestimmt man die Anzahl der Einträge im Feld einer Eigenschaft:

```
Möhre.#Vokabular == 3
Möhre.Vokabular == 'möhre'
Möhre.Vokabular[0] == 'möhre'
Möhre.Vokabular[1] == 'karotte'
Möhre.Vokabular[2] == 'mohrrübe'
```

Auf dieselbe Weise kann man die Einträge auch verändern. Die Länge dieses Feldes kann sich nicht ändern, die Einträge schon. Eine Anweisung wie

```
Möhre.#Vokabular = 4;
```

ist daher nicht erlaubt, und folgende Zuweisung

```
Möhre.Vokabular[3] = 'möhrchen';
```

schreibt den Wert an eine nicht definierte Position und korrumpiert sehr wahrscheinlich die Spieldaten. Es ist die Aufgabe des Spielers, sicherzustellen, dass so etwas nicht passiert. Will man im Laufe des Spiels Daten zu einem Eigenschaftsfeld hinzufügen, so muss man bei der Definition Platzhalter einsetzen, die man dann später überschreiben kann:

```
Vokabular = 'möhre', 'karotte', 'mohrrübe', '', '', '', '';
```

Manche Eigenschaften werden erst während des Spiels belegt und bestehen zu Beginn nur aus Platzhaltern:

```
Code = 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
```

Das ist unübersichtlich, daher kann man hier eine abkürzende Schreibweise verwenden, bei der die Anzahl der Einträge ähnlich wie bei der Definition von Feldern in eckigen Klammern angegeben wird:

```
Code[12];
```

× Die Felder von Eigenschaften haben immer 16-Bit-Einträge, und es können maximal 32 Einträge pro Eigenschaft definiert werden. Manchmal möchte man aber nur Bytes abspeichern, zum Beispiel bei Zeichenketten. Dies kann man mit dem Operator `->` machen, dann hat man 64 Bytes zur Verfügung:

```
Code = 'g'*$100 + 'a', 'n'*$100 + 'y', 'm'*$100 + 'e',
    'd'*$100, 0, 0, 0, 0;

Obj.Code->0 == 'g';
Obj.Code->1 == 'a';
Obj.Code->2 == 'n';
...
Obj.Code->15 == 0;
```

× Mit dem Operator `.&` kann man eine Eigenschaft eines Objektes in ein Feld umwandeln. Das ist in den meisten Fällen redundant, `obj.p[0]` und `obj.&p[0]` bewirken dasselbe. Wenn man das Feld jedoch einer Variable zuweisen will oder es an eine Routine übergeben will, muss man den Operator `.&` benutzen:

```
x = Möhre.&Vokabular;
vokabel(x[0]);
```

2.5 Klassen

Klassen sind ähnlich wie Objekte. Sie dienen anderen Objekten als Vorlage, tauchen aber selbst im Spiel nicht auf. Beim Erzeugen eines Objekts aus einer Klasse werden die Eigenschaften und die Attribute der Klasse auf das Objekt übertragen. Man sagt, die Eigenschaften der Klasse werden *vererbt*. Das neu erzeugte Objekt heißt dann *Instanz* der Klasse.

Das ist praktisch, wenn es mehrere Objekte gibt, die ein ähnliches oder gleiches Verhalten haben. Zum Beispiel könnte man eine Klasse `Nahrung` definieren:

```
Klasse Nahrung {
    ist essbar;
    Vitamine = 2;
    Kohlenhydrate = 2;
    Eiweiß = 2;
    Nährwert = {
        wenn (selbst ist verfault) antworte -1;
        antworte selbst.Vitamine +
            selbst.Kohlenhydrate +
            selbst.Eiweiß;
    }
```

```
}
```

Die Definition einer Klasse ist ähnlich wie bei Objekten. Unterschiedlich ist, dass Klassen unbedingt einen Bezeichner haben müssen und dass man den Ort einer Klasse nicht angeben kann. Die Klasse beschreibt das „Konzept“ eines Nahrungsmittels, das verschiedene Regeln enthält, aber keinem Ort zugewiesen ist. Die Definitionen innerhalb des Objektkörpers zwischen den geschweiften Klammern sind bei Klassen genau wie bei Objekten.

Mit dieser Definition kann nun eine Instanz der Klasse `Nahrung` erzeugt werden:

```
Nahrung Schwarzbrot;
```

Damit gibt es nun ein Objekt `Schwarzbrot`, das die Eigenschaften der Klasse `Nahrung` besitzt. Die Eigenschaften von `Schwarzbrot` können sich natürlich während des Spiels ändern. Der Bezeichner der Klasse wird hier als Direktive verstanden und bedeutet „erzeuge eine Instanz dieser Klasse“. Diese Schreibweise ist eine Abkürzung für:

```
Objekt Schwarzbrot {
    beerbt Nahrung;
}
```

Die Klassenzugehörigkeit kann also auch nach dem Schlüsselwort `beerbt` im Objektkörper angegeben werden.

Nicht alle Nahrungsmittel sind gleich, und deshalb kann man die geerbten Eigenschaften in der Definition der Instanzen ändern:

```
Nahrung Limette {
    Vitamine = 6;
}

Nahrung Schimmelkäse {
    Nährwert = 2;
    Kohlenhydrate = 0;
    ist verfault;
}

Nahrung Schokopudding {
    Kohlenhydrate = 4;
    Vitamine = 0;
}
```

Damit gibt es nun vier Nahrungsmittel mit folgenden Eigenschaften:

```
               Vitamine   Kohlenh.   Eiweiß   Nährwert
Schwarzbrot       2          2         2         6
Limette           6          2         2        10
```

```
Schimmelkäse     2          0          2          2
Schokopudding    0          4          2          6
```

Bei der Definition der Eigenschaft `Nährwert` der Klasse `Nahrung` tauchte eine besondere Variable auf: `selbst`. Diese Variable bezieht sich auf das Objekt, für das die Routine gerade ausgewertet wird. Dies ist notwendig, damit die Routine für alle Instanzen der Klasse gültig ist.

In diesem Beispiel werden die Eigenschaften ersetzt, dies ist der Regelfall. Neben der *ersetzenden* Vererbung können Eigenschaften aber auch *erweiternd* vererbt werden. Dazu muss eine Eigenschaft als *erweiternd* oder *additiv* deklariert werden. Dies geschieht mit der Angabe „+“ bei der Definition der Eigenschaft:

```
Eigenschaft + Vokabular;
```

Dann wird ein Feld von Eigenschaften definiert, dem die Einträge aus der Klassendefinition angehängt werden:

```
Klasse Münze {
    Vokabular = 'goldstück', 'münze';
}

Münze Dublone {
    Vokabular = 'dublone';
}

Dublone.#Vokabular == 3;
Dublone.Vokabular[0] == 'dublone';
Dublone.Vokabular[1] == 'goldstück';
Dublone.Vokabular[2] == 'münze';
```

. . . . .

Ein Objekt kann Instanz mehrerer Klassen sein. Zum Beispiel:

```
Objekt Trüffel {
    beerbt Nahrung, Rarität;
}
```

Hier werden zuerst alle Eigenschaften der `Nahrung` übertragen, dann die der `Rarität`, wobei eventuell bereits geerbte Eigenschaften wieder überschrieben werden. Wenn eine Eigenschaft also in beiden Klassen definiert ist, wird die der `Rarität` vererbt. Der `Trüffel` besitzt alle Attribute, die bei der `Nahrung` oder der `Rarität` (oder beiden) gesetzt sind.

Schließlich können auch Klassen von anderen Klassen erben:

```
Klasse Obst {
    beerbt Nahrung;
    Nährwert = {
        antworte 3*selbst.Vitamine;
    }
}
```

Dann ist `Obst` eine „Unterklasse“ von `Nahrung`. Eine Instanz von `Obst` erbt natürlich alle Eigenschaften von `Nahrung`, die nicht in der Definition von `Obst` überschrieben wurden.

Ob ein Objekt eine Instanz einer Klasse ist, kann man mit folgender Bedingung prüfen:

(*Objekt* `beerbt` *Klasse*)

In dieser Bedingung sind Unterklassen berücksichtigt. Wenn es also ein Objekt `Birne` der Klasse `Obst` gibt, sind die folgenden Bedingungen wahr:

```
(Birne beerbt Obst)
(Birne beerbt Nahrung)
```

2.6 Verschiedene Datentypen

Variablen und Eigenschaften von Objekten können nicht nur Zahlenwerte annehmen, sondern verschiedene Arten von Daten enthalten, wie z.B. Texte in Anführungsstrichen, Objekte, Klassen, Vokabeln in Hochkommas, Attribute, Eigenschaften oder Feldbezeichner. Intern werden diese Daten natürlich immer als Zahlen gespeichert. So ist ein Text nichts anderes als die Adresse im Speicher, an der er beginnt.

Vier Datentypen sind von besonderer Bedeutung: Objekte, Klassen, Routinen und Texte. Um zu erfahren, welchen dieser Typen eine Variable speichert, kann man die Systemfunktion `Datentyp` benutzen. Sie gibt einen der folgenden Werte zurück:

`Objekt`	ein Objekt
`Klasse`	ein Objektklasse
`Routine`	eine Routine
`Textstring`	ein Text in Anführungszeichen
`nichts`	keiner dieser Typen

Der Wert `nichts` bedeutet, dass in dieser Variable ein Wert gespeichert ist, der keinem der anderen vier Datentypen entspricht. Es ist meist ein Zahlenwert, kann aber auch z.B. eine Vokabel sein.

Dazu ein kurzes Beispiel:

```
Objekt Ding in Gegenstand {
    alpha = -1;
    beta = "Sic transit gloria mundi";
    gamma = {
        antworte zufall(0, 8, 15);
    }
    delta = Gegenstand;
    epsilon = 'epsilon';
}
```

Die Eigenschaften von `Ding` haben dann folgende Datentypen:

```
Datentyp(Ding.alpha) == nichts
Datentyp(Ding.beta) == Textstring
Datentyp(Ding.gamma) == Routine
Datentyp(Ding.delta) == Objekt
Datentyp(Ding.epsilon) == nichts
Datentyp(Ding) == Objekt
```

`Ding` selbst ist natürlich ein Objekt.

!! Die Funktion `Datentyp` unterscheidet nur die vier Hauptdatentypen und bezeichent alles andere als `nichts` (oder null). Daher ist etwas Vorsicht geboten, zum Beispiel kann folgendes gelten:

```
Datentyp(-3) == nichts
Datentyp(3) == Objekt
```

Dies ist so, wenn es ein Objekt mit der Nummer drei gibt. ZETA kann nicht zwischen der Zahl drei und dem Objekt drei unterscheiden. Die Funktion `Datentyp` schaut also nur, ob der Inhalt einer Variable im Bereich der Objekte, Klassen, Routinen oder Strings liegt und einer dieser Datentypen sein *könnte*.

2.7 Mitteilungen an Objekte

Objekte können miteinander kommunizieren, indem sie sich so genannte *Mitteilungen* (engl. *messages*) senden. Eine Mitteilung hat einen Absender und einen Empfänger. Außerdem kann sie – wie bei Routinen – Argumente übermitteln und eine Antwort zurückbekommen. Daher sieht die Syntax für eine Mitteilung auch ähnlich aus wie bei Routinen:

```
x = Computer.berechne(2, Nummer);
```

sendet die Mitteilung `berechne` an den `Computer` mit den Argumenten `2` und `Nummer` und speichert die Antwort in der Variablen `x`. Die runden Klammern müssen angegeben werden, auch, wenn keine Argumente übergeben werden. (Ohne Klammern betrachtet Zeta den Ausdruck einfach als die Adresse im Speicher, an der die Eigenschaft `Computer.berechne` steht.)

Der Empfänger, also der `Computer`, muss die Eigenschaft `berechne` besitzen. Sie könnte z.B. so aussehen:

```
Objekt Computer {
    berechne(x, y) = { antworte x*y; }
}
```

Dann würde die Mitteilung oben die Routine `berechne` aufrufen und den Wert `2*Nummer` in `x` ablegen.

Mitteilungen werden meistens an Eigenschaften vom Datentyp Routine gesendet, sie können aber an beliebige Eigenschaften geschickt werden. Je nach Datentyp passiert folgendes:

Eine `Routine` wird mit den angegebenen Argumenten aufgerufen, die Antwort ist der Rückgabewert der Routine. Ein `Textstring` wird ausgegeben, die Antwort ist `wahr`. Bei den Datentypen `Objekt`, `Klasse` oder `nichts` passiert nichts, die Antwort ist die Nummer des Objekts oder der Klasse, bzw. der Wert der Eigenschaft.

Die Übergabe von Argumenten hat nur bei Routinen eine Bedeutung. Argumente können auch an Eigenschaften von anderen Datentypen übergeben werden, aber sie werden nicht beachtet.

Dass verschiedene Datentypen Mitteilungen erhalten können, ist für die Adventure-Bibliothek von Vorteil, da Eigenschaften nicht immer vom selben Typ sein müssen:

```
Objekt Läufer {
    Beschreibung = "Der Läufer ist schön gewebt."
    ...
}

Objekt Vase {
    Beschreibung = {
        Wenn (selbst ist kaputt) Stop "Was früher
            eine wertvolle Vase war, ist nun ein
            Haufen Scherben."
        Stop "Eine schöne Ming-Vase."
    }
    ...
}
```

Im Zusammenhang mit der Kommunikation zwischen Objekten sind zwei Variablen von Bedeutung: `selbst` und `Sender`. Wie schon bei den Klassen kurz beschrieben, enthält `selbst` das Objekt, an das die Mitteilung geschickt wird. Der `Sender` ist das Objekt, das die Mitteilung gesendet hat. `Sender` kann null sein, wenn die Mitteilung aus einer Routine, die nicht zu einem Objekt gehört, z.B. dem Hauptprogramm, abgeschickt wurde.

Wenn die Eigenschaft, die die Mitteilung erhält, mehrere Einträge enthält, so werden diese der Reihe nach abgearbeitet, solange der Rückgabewert null oder `falsch` ist. Dies ist vor allem bei addtitiven Routinen, die geerbt werden, von Bedeutung: Wenn die Routine der Instanz nicht „eingreift“, also `falsch` zurückgibt, so wird die von der Klasse geerbte Routine aufgerufen, wenn es sie gibt. Die Antwort der Mitteilung ist immer die Antwort des zuletzt abgearbeiteten Eintrags.

. . . . .

Auch Klassen können Mitteilungen erhalten. Dies geht aber nur, wenn sie von einer ihrer Instanzen gesendet werden. Wenn `selbst` beim Senden der Mitteilung keine Instanz der Klasse, an die die Mitteilung gesendet wird, ist, passiert nichts.

Wenn eine Klasse eine Mitteilung erhält, behalten die Variablen `selbst` und `Sender` ihre Werte, so dass in einer Routine, die zur Klasse gehört `selbst` immer noch die Instanz und nicht die Klasse bedeutet.

Mitteilungen an Klassen werden häufig verwendet, wenn eine (nicht additive) Routine für eine Instanz nur leicht von der geerbten abweicht:

```
Klasse Nahrung {
    Verzehr = {
        wenn (!Hunger) stop "Du bist satt."
        "Du isst ", den(selbst);
        wenn (Hunger < selbst.Nährwert) {
            Hunger = 0;
            " und bist pappsatt.";
        } sonst {
            Hunger −= selbst.Nährwert;
            " und fühlst dich weniger hungrig.";
        }
    }
    Nährwert = 5;
}

Nahrung Weißbier {
    Verzehr = {
        wenn (Stunde < 16) stop "Nicht vor vier!";
        Nahrung.Verzehr();
    }
```

```
}
```

Die Mitteilung `Nahrung.Verzehr()` bewirkt, dass sich das Bier nach der Überprüfung der Uhrzeit so verhält, als ob es die Eigenschaft `Verzehr` geerbt hätte. Wenn viele Objekte die Eigenschaft Verzehr haben, ist es aber wohl einfacher, diese Eigenschaft als additive gemeinsame Eigenschaft zu deklarieren, und das wird auch in der Bibliothek gemacht.

2.8 Erzeugen und Vernichten von Objekten

ZETA bietet die – wenn auch eingeschränkte – Möglichkeit, Objekte während des Spiels zu erzeugen. Die Ressourcen der *Z*-Maschine sind jedoch beschränkt, und man kann Objekte nicht beliebig generieren, wie es in vielen objektorientierten Sprachen möglich ist. Damit ein Objekt erzeugt werden kann, muss das Objekt Instanz einer Klasse sein und ZETA muss wissen, wieviele Objekte erzeugt werden können. Dazu gibt man die maximal erzeugbaren Instanzen einer Klasse nach dem Klassenbezeichner in runden Klammern an:

```
Klasse Münze (24) {
    Name = "Münze";
    Wert = 1;
    Beschreibung = {
        "Eine Münze im Wert von ", num(selbst.Wert),
        " Zetmid."
    }
}
```

Nun werden vierundzwanzig Objekte reserviert, die dem Spieler und dem Programmierer zunächst nicht zugänglich sind, und die mit der Anweisung `m = erzeuge(Münze)` „erzeugt“ werden können. Diese Systemfunktion gibt ein erzeugtes Objekt zurück oder `nichts`, wenn keine Objekte mehr erzeugt werden konnten, da bereits alle vierundzwanzig Münzen im Spiel sind. Mit `zerstöre(m)` kann man eine Münze wieder aus dem Spiel nehmen, um sie dann eventuell später wieder erzeugen zu können.

Beim Erzeugen muss eine Klasse angegeben werden, da nur Instanzen von Klassen erzeugt werden können. Beim Zerstören muss ein Objekt angegeben werden, da immer eine konkrete Instanz zerstöre wird. Man kann auch Objekte erzeugen, die zu keiner Klasse gehören. Dann ist dieses Objekt für immer verloren, es kann nicht wieder erzeugt werden.

Nach dem Erzeugen befindet sich ein Objekt in `nichts` und es hat die Eigenschaften der Klasse. Diese Eigenschaften können natürlich geändert werden, wenn das Objekt erzeugt wurde:

```
x = erzeuge(Münze);
```

```
verschiebe x nach Spieler;
x.Wert = Zufall(1, 2, 5, 10);
```

Dies kann man eleganter beim Objekt selbst mit der Eigenschaft `erzeugen` angeben, da das erzeugte Objekt die Mitteilung `x.erzeugen()` erhält. Diese Eigenschaft kann, wie üblich, bis zu drei Argumente übergeben:

```
Klasse Münze (24) {
    ...
    erzeugen = {
        lokal w, o;
        selbst.Wert = w;
        verschiebe selbst nach o;
    }
}
```

Nun kann man mit `erzeuge(Münze, 50, Geldbeutel)` ein 50-Zetmid-Stück in seinen Geldbeutel befördern. Analog dazu gibt es die Eigenschaft `zerstören`, die beim Zerstören des Objekts aufgerufen wird:

```
Klasse Gigolo (16) {
    Name;
    erzeugen = {
        lokal r;
        selbst.Name = Zufall("Giovanni", "Luigi",
            "Carlo", "Luca", "Giuseppe", "Francesco");
        r = erzeuge(Rose);
        wenn (r) verschiebe Rose nach selbst;
    }
    zerstören = {
        lokal x = Tochter(selbst), y;
        solange (x) {
            y = Schwester(x);
            wenn (x beerbt Rose) zerstöre(x);
            x = y;
        }
    }
}
```

Wieviele Instanzen einer Klasse noch erzeugt werden können, kann man mit der Systemfunktion `verbleibend(k)` ermitteln. Zu Beginn des Spiels wäre `verbleibend(Gigolo)` sechzehn. Wenn dieser wert null ist, gibt ein weiteres `erzeuge(k)` anstelle eines Objekts `nichts` zurück.

3 Systemfunktionen und Besonderheiten

3.1 Eingabe von Text

Ein Textadventure muss natürlich die Eingabe von Text ermöglichen, um es spielbar zu machen, und selbst einfachere Programme benötigen ab und zu die Kommunikation mit dem Benutzer.

Die einfachste Art der Interaktion ist der einfache Tastendruck, der mit der Funktion `Taste()` abgefragt wird. Diese Funktion wartet solange, bis eine Taste gedrückt wird und gibt den ZSCII-Code der gedrückten Taste zurück:

```
Routine JaOderNein {
    lokal t;
    "Bitte antworten Sie mit (j)a oder (n)ein. ";
    schleife (;;) {
        t = Taste();
        wenn (t=='j') antworte wahr;
        wenn (t=='n') antworte falsch;
    }
}
```

Die Funktion `Taste` schreibt nichts auf den Bildschirm, das muss der Programmierer machen, wenn er es wünscht.

Man kann auch ganze Textzeilen einlesen. Dazu gibt es die Funktion `lese(Zeile, max)`, die die eingelesenen Zeichen der Zeile auf den String `Zeile` schreibt. Der Spieler kann nicht mehr als `max` Zeichen eingeben. Die Textzeile wird folgendermaßen auf `Zeile` geschrieben: In `Zeile[0]` steht die maximale Anzahl der gelesenen Zeichen, dieser Wert wird von `lese` auf `max` gesetzt. In `Zeile[1]` wird die Anzahl der tatsächlich gelesenen Zeichen abgelegt. Die ZSCII-Codes der Zeichen stehen in den Feldeinträgen ab `Zeile[2]`. Beim Einlesen werden alle Zeichen in Kleinbuchstaben umgewandelt.

Wenn der Spieler „Karl, n“ eingibt, sehen die Einträge in `Zeile` so aus:

```
Zeile[0] == max             Zeile[5] == 'l'
Zeile[1] == 7               Zeile[6] == ','
Zeile[2] == 'k'             Zeile[7] == ' '
Zeile[3] == 'a'             Zeile[8] == 'n'
Zeile[4] == 'r'             ...
```

Das Feld `Zeile` muss für mindestens `max + 2` Bytes Platz haben, besser noch für `max +`

3, da einige Interpreter eine Null an den eingegebenen Text anhängen.

Der eingelesene Text kann dann in Wörter aufgeteilt werden. Die Funktion dazu ist `splitte(Zeile, Wort, max)`. Der Text im Feld `Zeile` wird in (maximal `max`) Wörter „aufgesplittet“, die Wörter werden auf dem Feld `Wort` abgelegt.

`Wort->0` hat den Wert `max`, `Wort->1` enthält die Anzahl der gefundenen Wörter. Danach kommen die Einträge, zu jedem Wort vier Bytes. Die ersten beiden Bytes enthalten die Adresse der Vokabel, wenn das Wort im Wörterbuch des Spiels steht, oder null, wenn das Wort nicht bekannt ist. Das dritte Byte gibt die Position des ersten Zeichens des Worts in `Zeile` an, das vierte Byte ist die Länge des Worts.

Einige Sonderzeichen, wie das Komma, werden als einzelnes Wort betrachtet. Wenn man die Wörter von eins bis `max` durchnummeriert, fängt jeder Vier-Byte-Block bei `Wort->(n*4-2)` bzw. bei `Wort-->(n*2-1)` an. Die oben stehende Belegung von `Zeile` würde nach dem Splitten folgende Einträge in `Wort` ergeben:

```
Wort->0 == max
Wort->1 == 3

Wort-->(2/2) == 'karl'      Wort->4 == 2
                            Wort->5 == 4

Wort-->(6/2) == ',\'        Wort->8 == 9
                            Wort->9 == 1

Wort-->(10/2) == 'n\'       Wort->12 == 8
                            Wort->13 == 1
```

Hier ist zu beachten, dass in `Wort` lange und kurze Einträge gemischt auftreten. Deshalb benutzt man auch besser die Operatoren `-->` und `->` anstatt der eckigen Klammern.

Während des Kompilierens legt ZETA ein Wörterbuch an, in das alle Vokablen, die im Laufe des Spiels zwischen Hochkommas definiert werden, abgelegt werden. Dieses Wörterbuch wird dann beim Aufteilen der Textzeile zu Rate gezogen, und ein Wort mit der passenden Vokabel belegt, wenn sie bekannt ist.

Die Textanalyse, die von `splitte` durchgeführt wird, ist nur sehr einfach. Wenn das Spiel z.B. die Vokabel `'rot'` kennt, kennt es die gebeugte Form `'rotes'` noch lange nicht. Diese eingehendere Analyse macht der Parser der Adventure-Bibliothek.

3.2 Textkanäle

Ein- und Ausgabetexte werden durch so genannte *Textkanäle* in den Interpreter hinein und aus ihm hinaus geleitet. Normalerweise ist der Eingabekanal die Tastatur, der Ausgabekanal ist der Bildschirm. Dies kann man aber ändern.

Die Funktion `Eingabekanal(x)` wählt den aktiven Eingabekanal aus. Eingabekanal 0 ist die Tastatur, Eingabekanal 1 ist eine Datei. Andere Eingabekanäle gibt es nicht, und es ist immer nur einer dieser Kanäle aktiv. Wenn der Eingabekanal auf eins geschaltet wird, fragt der Interpreter nach einer Datei, dies ist nicht Aufgabe des Programms.

Mit der Funktion `Ausgabekanal(x)` können Ausgabekanäle aktiviert (`x>0`) und deaktiviert (`x<0`) werden. Es gibt vier Ausgabekanäle:

`1`	den Bildschirm
`2`	ein Protokoll des gesamten Texts (ohne Statuszeile)
`3`	ein String im Speicher
`4`	ein Protokoll der eingegebenen Befehle

Anders als bei der Eingabe können hier mehrere Kanäle aktiv sein.

Ausgabekanal 3 sollte nicht mit `Ausgabekanal` aktiviert und deaktiviert werden. Stattdessen gibt es eine weitere Funktion, `Textumleitung(f)`, die allen ausgegebenen Text auf das String-Feld `f` umleitet, bis diese Umleitung mit `Textumleitung(0)` wieder ausgeschaltet wird. In den ersten beiden Bytes `f-->0` steht die Anzahl der Zeichen, ab `f->2` steht der Text. Diese Funktion kann z.B. benutzt werden, um die Länge eines Textes zu ermitteln oder um Texte vor der Ausgabe zu formatieren.

!! Hierbei muss man aufpassen, dass das Feld `f` genug Einträge zur Verfügung hat, um den Text aufzumehmen. Wenn die Textumleitung nicht rechtzeitig ausgeschaltet wird, läuft das Feld über und korrumpiert wahrscheinlich andere Daten.

3.3 Schrift und Farbe

Auf manchen Interpretern kann Schrift in verschiedenen Schriftstilen und -farben angezeigt werden.

Mit `Schriftstil(Stil)` kann man den nachfolgenden Text mit einer der folgenden Textauszeichnungen versehen:

`0`	Normale Textschrift
`kursiv`	kursive oder schräggestellte Schrift
`fett`	Fettdruck
`invers`	Vorder- und Hintergrundfarbe vertauscht
`monospace`	diktengleiche Schreibmaschinenschrift

Diese Schriftstile können auf manchen Interpretern (z.B. *Frotz*) kombiniert verwendet werden, indem man die Funktion `Schriftstil` mehrmals aufruft, z.B.:

```
Schriftstil(Monospace);
Schriftstil(invers);
"   Betreten   \n
\    streng    \n
\   verboten   ";
Schriftstil(0);
```

Mit `Schriftstil(0)` setzt man den Stil immer auf die normale Textschrift zurück. (Die Backslashs am Anfang der zweiten und dritten Textzeile, denen ein Leerzeichen folgt, verhindern, dass der *white space* bis an die eigentlichen Wörter wie üblich ignoriert wird, damit das Layout des Schilds nicht zerstört wird.)

Diese Kombination wird nicht von jedem Interpreter verstanden, und nicht jeder Interpreter verfügt über verschiedene Schriftarten. Manche Interpreter stellen kursive Schrift unterstrichen und Fettdruck in einer hervorgehobenen Farbe dar. Am besten werden Textauszeichnungen sparsam verwendet und Kombinationen von Schriftstilen vermieden. (Eine Regel, die im übrigen auch für gedruckten Text gilt.)

. . . . .

Einige Interpreter können farbigen Text darstellen. Dann kann man mit der Funktion `Textfarbe(VG, HG)` die Vorder- und Hintergrundfarbe des Texts ändern. Folgende Farben stehen zur Verfügung:

`0`	*momentan benutzte Farbe*
`1`	*standardmäßig benutzte Farbe*
`2`	schwarz
`3`	dunkelrot
`4`	grün
`5`	gelb oder braun
`6`	dunkelblau
`7`	violett
`8`	türkis
`9`	weiß oder hellgrau

Bevor man diese Funktion aufruft, sollte man sicherstellen, dass der Interpreter über die Möglichkeit, die Textfarbe zu ändern, verfügt (und dass der Spieler die Option „farbiger Text“ nicht ausgeschaltet hat). Wenn der Interpreter Farbe darstellen kann, wird ein Bit im *Header*, im „Kopf“ der Spieldatei, gesetzt und folgende Bedingung ist wahr:

```
((0->1) & 1 != 0)
```

Die Farbangabe `0` bereitet bei einigen Interpretern, z.B. *Zip*, Probleme, deshalb sollte man sie nicht verwenden. Farbiger Text ist nicht jedermanns Sache und man sollte dem Spieler die Möglichkeit geben, ihn abzustellen.

3.4 Der Bildschirm

Der „Bildschirm“ ist der Darstellungsbereich des Interpreters, der meistens nicht der physikalische Bildschirm ist, sondern ein Anwendungsfenster. Der Bildschirm der *Z*-Maschine teilt sich in zwei Bereiche: Das untere oder *Textfenster* und das obere oder *Statusfenster*. Das Textfenster wird immer dargestellt, das Statusfenster kann ein- und ausgeblendet werden. Dies geschieht mit der Systemfunktion

```
Statusfenster(Z);
```

Wenn `Z` null ist, wird das Statusfenster ausgeblendet, ein positives `Z` blendet ein Statusfenster am oberen Rand ein, das `Z` Zeilen hoch ist. Das Statusfenster wird zwar eingeblendet, ist aber erst sichtbar, wenn etwas in das Fenster geschrieben wird. Vor allem wird das Fenster nicht automatisch invers, d.h. mit umgekehrten Farben, dargestellt, wie man es aus den meisten Adventures gewohnt ist. Mit der Funktion

```
aktivesFenster(F);
```

kann man das Textfenster (`F==0`) oder das Statusfenster (`F==1`) für die Ausgabe von Text auswählen.

Das Statusfenster ist dem Textfenster überlagert. An Stellen, an denen im Statusfenster kein Text steht, sieht man das, was im Textfenster steht. Damit das Textfenster nicht „durchscheint“, wird das Statusfenster daher vor dem Beschreiben mit inversen Leerzeichen gefüllt.

Im Statusfenster wird nicht gescrollt, es ist nicht erlaubt, in Zeilen zu schreiben, außerhalb des mit `Statusfenster` definierten Bereichs liegen. Der Text im Statusfenster wird immer in einer Schreibmaschinenschrift geschrieben, deren Zeichen dieselbe Breite haben. Wenn man die Größe dieser Zeichen als Einheit benutzt, sind die Abmaße des gesamten Bildschirms:

```
h = bsHöhe();
b = bsBreite();
```

Wenn das Statusfenster aktiv ist, kann man die Position, an der als nächstes geschrieben werden soll, den *Cursor*, ändern mit der Systemfunktion

```
cursorPos(x, y);
```

Die obere linke Ecke des Statusfensters ist `(1, 1)`, die untere rechte Ecke hat die Koordinaten `(bsBreite(), Z)`. Mit den Funktionen

```
x = cursorX();
y = cursorY();
```

kann man die momentane Position des Cursors abfragen. Schließlich kann man mit

```
löscheFenster(F);
```

den Inhalt des Fensters `F` löschen. Wenn `F==-1` ist, wird der gesamte Bildschirm gelöscht, das Statusfenster wird ausgeblendet und der Cursor im Textfenster wird in die obere linke Ecke gesetzt. Dies ist der Zustand, in dem sich der Bildschirm zu Beginn des Spiels befindet.

. . . . .

Der Interpreter bricht immer ganze Wörter um, wenn er Text ausgibt. Auf diese Weise ist der Text gut zu lesen. Gerade im Statusfenster ist es aber manchmal praktisch, diese Art der Texttrennung aufzuheben und Text immer am rechten Bildschirmrand umzubrechen. Mit der Funktion

```
Wortumbruch(Flagge);
```

kann man das Umbrechen ganzer Wörter daher für das momentan aktive Fenster ein- (`Flagge==0`) und ausschalten (`Flagge==1`).

3.5 Laden und Speichern

Der Zustand der *Z*-Maschine ändert sich ständig. Man kann ihn auf eine externe Datei schreiben, um ihn später wieder einlesen zu können. Dazu stehen in ZETA die Funktionen

```
e = speichere();
e = lade();
```

zur Verfügung. Wenn `e` den Wert `falsch` hat, schlug das Speichern oder Laden des Spielstands fehl. Bei erfolgreichem Speichern wird `wahr` zurückgegeben, bei erfolgreichem Laden ist der Rückgabewert zwei, und das Programm wird an der Stelle nach dem entsprechenden Speicherbefehl fortgesetzt. In der Praxis heißt das, dass der Rückgabewert des Ladens egal ist, denn alles was nach `lade` steht wird nur ausgeführt, wenn das Wiederherstellen des Zustands nicht geklappt hat. `speichere` hat hingegen drei mögliche Rückgabewerte: Null für einen Fehler beim Speichern, eins für erfolgreiches Speichern und zwei für erfolgreiches Laden eines zuvor gespeicherten Zustands.

Die Frage nach einem Dateinamen wird immer vom Interpreter gestellt und kann ein einfacher Prompt sein oder ein Dialog. Der Autor hat darauf keinen Einfluss.

Anstatt den momentanen Zustand der *Z*-Maschine auf eine Datei zu schreiben, kann man ihn im Speicher des Rechners ablegen, und ihn dann später wieder wiederherstellen. Dies wird von der Adventure-Bibliothek gemacht, um die Daten für das Rückgängigmachen von Zügen zwischenzuspeichern. Die Funktionen heißen

```
e = speichereZug();
e = ladeZug();
```

Wenn ein Interpreter kein Zwischenspeichern von Spielständen im Arbeitsspeicher ermöglicht, ist das Resultat von `speichereZug()` gleich –1. Es ist nicht erlaubt, einen Zustand mit `ladeZug` vom Speicher zu lesen, wenn zuvor kein Zustand mit `schreibeZug` abgelegt wurde.

. . . . .

Es besteht die Möglichkeit, den Inhalt von Feldern auf Dateien zu schreiben oder sie von Dateien zu laden. Die Funktion

```
e = speichereFeld(f, n, Dateiname);
```

schreibt `n` Bytes des Feldes `f` auf die Datei mit den Namen `Dateiname`. `e` ist `falsch`, wenn es Probleme beim Abspeichern gab. Mit der Funktion

```
e = ladeFeld(f, n, Dateiname);
```

werden von der Datei `Dateiname` `n` Bytes gelesen und auf das Feld `f` geschrieben. `e` ist die Anzahl der erfolgreich gelesenen Bytes. Wenn `e` kleiner als `n` ist, gab es Probleme beim Einlesen.

Die Dateilänge `n` wird immer in Bytes angegeben. Normalerweise haben Feldeinträge in ZETA die eine Länge von zwei Bytes, so dass hier zweimal die Länge des Feldes angegeben

werden muss, wenn das Feld kein String ist.

Der `Dateiname` ist ein Feld vom Typ String, das den Namen, unter dem die Daten gespeichert werden sollen, enthält. Der Dateiname muss sich an das 8.3-Format halten. Das heißt er darf höchstens acht Zeichen lang sein, evetuell gefolgt von einem Punkt und einer Dateinamenerweiterung von maximal drei Zeichen. Wird keine Erweiterung angegeben, so ist die Erweiterung `.aux`.

Teil II:

Arbeiten mit dem Compiler

4 Der Compiler und seine Optionen

Der einfachste Weg, um eine Spieldatei zu erzeugen, ist der Aufruf von ZETA mit dem Namen des Quelltexts als einzigem Parameter:

```
zeta ganymed
```

Wenn keine Dateiendung für den Quelltext angegeben wird, hängt ZETA automatisch `.zet` an. Die erzeugte Spieldatei heißt `ganymed.z5`.

Die vollständige Syntax für das Erzeugen von Spieldateien mit ZETA ist:

`zeta` {*Optionen*} *Quelltext*{`.zet`} {*Spieldatei*{`.z5`}}

Die Angaben in geschweiften Klammern sind optional.

4.1 Optionen beim Aufruf des Compilers

Es gibt verschiedene Möglichkeiten, das Verhalten des Compilers ZETA zu beeinflussen. Die gebräuchlichste ist, ZETA mit einigen Optionen zu starten. Zum Beispiel schreibt

```
zeta –s smaug
```

eine kurze Statistik der erzeugten Spieldatei, wenn sie erfolgreich erstellt wurde. Viele Optionen dienen nur dazu, die Ausgabe während des Kompilierens zu steuern.

`-l` legt ein „Logbuch“ des Compiler-Outputs an: Alles, was auf dem Bildschirm ausgegeben wird, wird zusätzlich in der Datei `smaug.log` mitprotokolliert. Das ist nützlich, um Fehlermeldungen nachlesen zu können.

`-z` gibt eine Speichertabelle der erzeugten Datei aus, die nützlich sein kann, wenn man an die Grenzen der erlaubten Größen stößt, sonst wohl aber eher akademisch ist. Wer es gerne sehr ausführlich mag, kann sich mit `-y` eine Liste aller Symbole, das heißt aller globalen Bezeichner im Quelltext ausgeben lassen.

Mit `-w` wird die Ausgabe von Warnungen unterdrückt und `-o` listet die Objekte auf, wenn sie kompiliert werden. Die Option `-h` schließlich gibt eine Liste der möglichen Optionen aus, ohne ein Spiel zu generieren.

Andere Optionen haben größeren Einfluss auf die erzeugte Spieldatei. Mit `-g` kann man eine „großes“ Spiel erzeugen: Anstelle einer `z5`-Datei wird eine Datei der Version `z8` erzeugt, die Platz für doppelt so viele Daten bietet: 512 anstatt 256 Kilobytes.

Der „kritische Modus“ wird mit `-k` aktiviert und bricht den Kompilerungsvorgang beim ersten Fehler ab.

Optionen sind entweder gesetzt oder nicht. Die Angabe von Optionen beginnt mit einem Minus-Zeichen, gefolgt von beliebig vielen Zeichen. `-zsl` schreibt also ein Logbuch und zeigt die Statistik und die Speichertabelle des erzeugten Spiels an.

Man kann Optionen auch wieder ausschalten, indem man dem ihnen ein Ausrufezeichen voranstellt. Diese Verneinung gilt immer nur für die direkt folgende Option. Mit `-a!bc` werden die Optionen `a` und `c` aktiviert, die Option `b` wird ausgeschaltet.

Eine Liste der möglichen Optionen steht im Anhang.

. . . . .

Der Compiler belegt für die verschiedenen Bereiche der zu erzeugenden Spieldatei Speicher auf der Maschine. Alle Opcodes werden zum Beispiel auf ein großes Feld gelegt, alle Texte und alle Objektdaten genauso. Aus diesen temporären Datenblöcken wird dann die Spieldatei assembliert.

Die Größe einiger dieser Speicherbereiche ist variabel. So kann man mit der Option

```
$maxcode=200000
```

den Speicherbereich für die Opcodes, also den eigentlichen *Z*-Code, auf 200.000 Bytes setzen. Der Zusatz `k` erlaubt die Angabe des Speicherplatzes in Kilobyte. Die Angabe

```
$maxcode=200k
```

stellt 200 Kilobyte für den *Z*-Code bereit. Das entspricht 204.800 Bytes – ein Kilobyte hat 1.024, nicht 1.000 Bytes.

Diese Einstellungen werden meist nicht verändert, es sei denn es ist zuwenig Speicher auf der Maschine vorhanden oder der Quelltext ist so groß, dass der in einem Bereich bereitgestellte Speicher nicht ausreicht. In beiden Fällen wird dem Autor gesagt, wie er das Problem beheben kann.

Als letzter Ausweg können Daten, die während des Kompilierens benötigt werden, mit der Option `-t` auf temporäre Dateien ausgelagert werden. Die Dateien fungieren als Speicher und

werden wieder gelöscht, nachdem die Spieldatei erzeugt wurde.

Mit `-c` kann man einen „Trockenlauf“ starten: Nur die Compiler-Parameter werden ausgegeben.

Eine weitere Option ist die Angabe von Verzeichnis-Pfaden nach einem Plus-Zeichen (+). Dateien, die während des Kompilierens eingebunden werden sollen, werden in allen angegebenen Pfaden gesucht. Alle Pfade werden der Reihe nach abgearbeitet, der erste Treffer wird genommen. Der einzige Pfad, der bereits angegeben ist, ist das momentane Arbeitsverzeichnis.

Diese Angaben ermöglichen es, ein wenig Ordnung zu halten und verschiedene Bibliotheken bequem aus verschiedenen Verzeichnissen anzuziehen. Eine andere Anwendung wäre das Testen einer neuen Version einer Bibliothek. Statt `+qlib` gibt man dann `+qlib_beta` an, um die neuen Dateien ins Projekt einzubinden und deren Funktion zu überprüfen. (Das geht natürlich nur, wenn die Dateinamen in beiden Bibliotheken dieselben sind.)

4.2 Steuerdateien

In einem Projekt arbeitet man oft mit denselben Optionen. Um das Setzen von Optionen zu vereinfachen, ermöglicht es ZETA, so genannte Steuerdateien anzulegen. Diese Dateien haben die Endung `.zak` („ZETA-Ablaufkontrolle“) und enthalten Optionen, die genau wie beim Aufruf angegeben werden.

Eingebunden wird eine solche Datei mit einem vorangestellten Doppelpunkt. Zum Beispiel zieht

```
zeta :eldorado
```

die Steuerdatei `eldorado.zak` an.

In einer solchen Steuerdatei wird alles so interpretiert, als ob es auf der Kommandozeile beim Aufruf stünde. Anstelle von `:eldorado` könnte man also alle in Dieser datei angebenenen Optionen auch in den Aufruf schreiben. Leere Zeilen werden ignoriert, genauso wie alles, was nach einem Lattenzaun (#) steht. Eine typische Steuerdatei sähe so aus:

```
# Ragnarök, (C) 2003 Björn Magnusson

:standard                    # Standard-Optionen
```

```
+lib/zlib +lib/raglib      # Pfade für Bibliotheken
$maxcode=320k              # Mehr Speicher für Code
-g                         # z8-Datei erzeugen
-l -s -z                   # Logbuch und Statistik
rag_beta.zet               # Quellcode (Beta-Version)
ragnarok.z8                # Spieldatei
```

Steuerdateien können auch andere Steuerdateien anziehen, nur sollte man dafür sorgen, dass sich die Steuerdateien nicht selbst aufrufen.

× Statt einer Steuerdatei kann man natürlich auch eine *Batch*-Datei oder ein *Makefile* benutzen, in denen die bevorzugten Optionen gesetzt werden. Die Steuerdatei ist nur eine Alternative zur Angabe der Optionen auf der Kommandozeile des Betriebssystems.

4.3 Steueranweisungen im Quelltext

Einige Direktiven zur Steuerung des Kompilierungsvorgangs können auch im Quelltext angegeben werden. Diese Steueranweisungen beginnen mit einem Doppelkreuz, #, und müssen – im Gegensatz zu allen anderen Anweisungen – nicht mit einem Semikolon abgeschlossen werden. (Es schadet aber nicht, ein Semikolon zu setzen.)

Eine häufig gebrauchte Steueranweisung ist das Einfügen von Dateien:

```
#datei "Dateiname"
```

Die angegebene Datei wird dann an dieser Stelle geöffnet und eingelesen. Das ist so, als ob der Inhalt dieser Datei an dieser Stelle im Quelltext stünde. Das Gliedern des Quelltextes in verschiedene Dateien macht ein Projekt aber übersichtlicher. Und schließlich wird auch die Adventure-Bibliothek mit diesem Befehl in ein Spiel eingebunden.

Der angegebene Dateiname kann auch einen Pfad enthalten. Wenn ein absoluter Pfad angegeben wird, so wird diese Datei geöffnet. Ansonsten wird in den zuvor angegebenen Pfaden gesucht, mindestens jedoch im momentanen Arbeitsverzeichnis.

Zusätzlich zur Definition von Pfaden mit *+Pfadname* beim Aufruf kann man im Quelltext den Suchpfad noch erweitern mit:

```
#pfad "Pfadname"
```

Weiterhin gibt es eine Anzahl von Steuerdirektiven, die es dem Programmierer ermöglichen, Teile des Quelltextes nur unter bestimmten Bedingungen zu kompilieren. Diese Anweisungen sind der `wenn-sonst`-Anweisung ähnlich:

```
#wenn {def | ndef} Ausdruck
    Anweisungen
#sonst
    andere Anweisungen
#ende
```

Die Angabe von `#sonst` ist optional, `#ende` muss auf jeden Fall angegeben werden. Drei verschiedene Bedingungen können mit `#wenn` benutzt werden:

`#wenn def` *Name*
ist wahr, wenn im Quelltext etwas mit dem angegebenen Namen definiert wurde. Dabei speilt es keine Rolle, ob der Name ein Objekt, eine Variable, eine Routine oder etwas anderes bezeichnet, er muss lediglich bereits bekannt sein. Definitionen, die weiter hinten im Quelltext stehen, werden nicht berücksichtigt.

`#wenn ndef` *Name*
ist wahr, wenn der Name in Zeta noch nicht benutzt wurde.

`#wenn` *Ausdruck*
ist wahr, wenn der Ausdruck wahr ist. Der Ausdruck darf nur konstante Werte enthalten, da er während des Kompilierens ausgewertet werden muss. Unbekannte Bezeichner dürfen hier ebenfalls nicht verwendet werden. (In den anderen beiden Bedingungen natürlich schon.)

Diese Direktiven können auch innerhalb von Routinen oder Objektdefinitionen verwendet werden:

```
...
#wenn def Debug
    "Objekt Nr. ", num(o);
#sonst
    o.Name();
#ende
...
```

Mit diesen Steueranweisungen kann man auch ganze Blöcke im Quelltext deaktivieren, ohne mit eventuell bereits existierenden Kommentaren in Konflikt zu geraten:

```
#wenn def niemals
    ...
#ende
```

Das funktioniert allerdings nur dann, wenn `niemals` nicht verwendet wird.

× Etwas aufpassen muss man, wenn man Namen auf ihre Existenz übeprüft.

```
#wenn def Expertenmodus
```

ist wahr, wenn es den Bezeichner `Expertenmodus` in diesem Spiel gibt, wohingegen

```
#wenn Expertenmodus
```

wahr ist, wenn `Expertenmodus` einen Wert hat, der von null verschieden ist. Wenn es den Bezeichner `Expertenmodus` nicht gibt, erzeugt die zweite Anweisung eine Fehlermeldung.

. . . . .

Um besser verfolgen zu können, was gerade kompiliert wird, kann man Texte ausgeben, während ZETA den Quelltext abarbeitet:

`#text` {`Warnung` | `Fehler` | `Fatal`} "*Meldung*"

Wenn der Text der Anweisung `#text` direkt folgt, wird er einfach ausgegeben. Wenn eine weitere Angabe zum Fehlercode gemacht wird, behandelt ZETA den Text, so als ob er eine normale Fehlermeldung ist. Insbesondere bricht etwas wie

```
#wenn ndef Debug
#text Fatal "Das Modul R3 kann nur im Debug-Modus laufen"
#ende
```

den Kompiliervorgang sofort ab.

. . . . .

ZETA bietet die Möglichkeit, Konstanten, Objekte und Routinen „blind“ zu definieren:

`#blinddef` *Definitionstyp Bezeichner* {= *Wert*}

„Blinddefinition“ bedeutet, dass nichts definiert wird, wenn bereits etwas mit diesem *Bezeichner* definiert wurde. Ansonsten wird je nach *Definitionstyp* eine `Konstante` mit dem *Wert*, eine `Routine`, die den *Wert* zurückgibt oder ein `Objeklt` ohne Eigenschaften definiert. Objekte können keine Wertezuweisung haben. Die Definitionen

```
#blinddef Konstante max_Last = 50;
#blinddef Routine ist_schwer = wahr;
#blinddef Objekt Tragehilfe;
```

sind also Abkürzungen für:

```
#wenn ndef max_Last
    Konstante max_Last = 50;
#ende
#wenn ndef ist_schwer
    Routine ist_schwer() { antworte wahr; }
#ende
#wenn ndef Tragehilfe
    Objekt Tragehilfe;
#ende
```

Weiterhin kann man so genannte *Prototypen* definieren:

`#proto` *Definitionstyp Bezeichner*

Ein Prototyp macht in ZETA nichts anderes als dem Compiler zu sagen: Der *Bezeichner* steht für einen *Definitionstyp*, der eine `Routine`, ein `Objekt`, ein `Befehl` oder eine `Eigenschaft` sein kann. Normalerweise weiß ZETA, welchen Typ ein Bezeichner hat, auch wenn dieser noch nicht definiert wurde. Es macht eine Annahme aus dem Zusammenhang. So steht für ZETA ein unbekannter Bezeichner für eine Eigenschaft, wenn sie nach einem Punkt steht, für ein Objekt, wenn er vor einem Punkt steht. In manchen Fällen, besonders dann, wenn man den Bezeichner ohne offensichtlichen Kontext als Argument an eine Routine übergeben will, macht ZETA falsche Annahmen. Eine Möglichkeit, dies zu umgehen, ist, die Definition zu dem betreffenden Bezeichner vor die mehrdeutige Stelle zu platzieren. Dies kann aber manchmal umständlich sein. Eine einfache Möglichkeit, dem Compiler den Typ des Bezeichners mitzuteilen, ist, am Anfang des Quelltexts einen Prototyp zu definieren.

Eigenschaften, die mit einem Prototyp definiert werden, sind immer individuelle Eigenschaften. Gemeinsame Eigenschaften müssen mit `Eigenschaft` definiert werden.

. . . . .

Mit der Anweisung `#lib`, die als erste in einer eingebundenen Datei stehen sollte, kann man diese Datei als „Bibliotheksdatei“ deklarieren. Damit unterdrückt man alle Warnungen, die in dieser Datei auftreten um zu vermeiden, dass der Bildschirm mit Warnungen über nicht benutzte Routinen oder Objekteigenschaften überquillt. Fehler werden allerdings wie gewohnt angezeigt und führen zum Abbruch.

Ein weiterer Effekt einer Bibliotheksdatei ist, dass man Routinen, Objekte, Klassen und Befehle, die in ihnen definiert sind, ersetzen kann. Dazu muss man die Steueranweisung

`#ersetze` *Definitionstyp Bezeichner*

im Quelletxt platzieren, und zwar vor dem Einbinden der Bibliothek, die die zu ersetzenden Definition ethält. Damit weist man ZETA an, die Definition in der Bibliothek zu ignorieren. Natürlich muss man diese Definition nachreichen, dies kann man überall im Quelltext machen, es muss nur nach `#ersetze` erfolgen.

Um zum Beispiel Dunkelheit in einem m Spiel auszuschließen, kann man die Lib-Routine `Spendet_Licht` ersetzen:

```
#ersetze Routine Spendet_Licht
#datei "zlib.zin"

Routine Spendet_Licht(obj) {
    antworte (obj != nichts);
}
```

Mögliche *Definitionstypen* bei der Ersetzung sind `Objekt`, `Klasse`, `Befehl` und `Routine`. Systemfunktionen können bislang nicht ersetzt werden.

. . . . .

Mit `#alias` kann man Symbolen, das heißt Schlüsselwörtern, Operatoren und Bezeichnern in ZETA, neue Synonyme vergeben:

`#alias` *Synonym Symbol*

Das ist in der Regel nicht sehr nützlich, im Gegenteil: Die Eindeutigkeit der Bezeichner geht dadurch verloren. Einigen Leuten gefallen aber die deutschen Schlüsselwörter von ZETA nicht, die die Eleganz und Knappheit der aus anderen Programmiersprachen bekannten englischen *Keywords* entbehren. Dies kann man mit `#alias` ändern:

```
#alias if wenn
#alias else sonst
#alias while solange
#alias switch jenach
...
```

Hier werden keine Anführungszeichen gesetzt. Das Synonym darf noch nicht verwendet worden sein und das Symbol, zu dem ein Synonym definiert wird, muss bekannt sein.

× Die alten Symbole werden nicht ersetzt. ZETA versteht nun `if` und `wenn` und behandelt sie beide gleich: als Anweisung für eine Bedingung.

. . . . .

Der *Z*-Code erlaubt es, Text zu komprimieren, indem häufig wiederkehrende Textfragmente nur einmal abgelegt werden und im laufenden Text durch einen Platzhalter dargestellt werden. Eine Möglichkeit, dies zu tun, ist Kurzzstrings zu definieren, eine weitere sind Abkürzungen, die mit

```
#abk "Textfragment"
```

definiert werden können. Bis zu 64 solcher Abkürzungen können definiert werden, bevor der erste Text in Gänsefüßchen im Quelltext auftaucht. Nur wenn die Option `-a` gesetzt ist, werden diese Abkürzungen jedoch benutzt. Automatisch wird dann jeder Text auf die definierten Textfragmente hin untersucht, die durch Platzhalter ersetzt werden. Dies kann bis zu zehn Prozent Speicher einsparen, und wird meist nur benutzt, wenn die Größe der Spieldatei an ihre Grenzen stößt. Vielleicht ist es aber auch keine schlechte Idee, ein Spiel vor seiner Veröffentlichung zu komprimieren.

× Text wird im *Z*-Code in so genannten *Z*-Zeichen gespeichert, so dass Dreiergruppen jeweils zwei Bytes belegen. Das Leerzeichen sowie die Kleinbuchstaben (ohne Umlaute) belegen je ein *Z*-Zeichen. Großbuchstaben, der Zeilenumbruch, die Ziffern und die meisten Interpunktionen belegen je zwei *Z*-Zeichen und alle anderen Zeichen, darunter auch die deutschen Umlaute und das Eszett, benötigen vier *Z*-Zeichen. Der Platzhalter für eine Abkürzung belegt zwei *Z*-Zeichen. Abkürzungen wie „er“ oder „D“ kürzen also nichts ab und werden vom Compiler mit einer Warnung ignoriert. Die Abkürzung „ü“ hingegen spart zwei Zeichen pro Ersetzung.

Gute Beispiele für Abkürzungen sind kurze, oft auftretende Fragmente wie Endungen und Vorsilben („en “, „ ver“), Konjunktionen, Artikel und Präpositionen („ und “, „Der “), Kombinationen von Konsonanten und Umlauten („schw“, „ße“) und Leer- und Satzzeichen („, “) sowie Kombinationen daraus („, dass“). Der Bibliothek liegt eine Liste guter Abkürzungen bei.

Die Effizienz der gewählten Abkürzungen wird ausgegeben, wenn man zusätzlich zur Option `-a` die Statistik-Option `-s` benutzt.

▪ ▪ ▪ ▪ ▪

Schließlich gibt es noch zwei Steueranweisungen, mit denen man die Seriennummer der Spieldatei und die *Release*, die laufende Nummer der Veröffentlichung, angeben kann:

```
#release Release
#seriennr Seriennummer
```

Die *Release* ist eine Zahl, die Seriennummer eine Folge von nicht mehr als sechs Zeichen, die in der Regel nur Ziffern enthält.

Jede Spieldatei hat eine Release und eine Seriennummer, und wenn der Autor sie nicht explizit angibt, so ist die Release eins und die Seriennummer das Datum des Kompilierens im Format `jjmmdd`, also `020501` für den ersten Mai 2002.

5 Fehler suchen und beseitigen

5.1 Fehler beim Kompilieren

Wenn ZETA beim Kompilieren Fehler findet, wird eine Fehlermeldung ausgegeben, die den Fehlergrad, die Stelle im Quelltext, an der der Fehler aufgetreten ist und eine Beschreibung des Fehlers angibt. Zum Beispiel zeigt

```
Fehler (modul_1.zet, 12): Hier wird ein Operator erwartet.
```

an, dass in Zeile 12 der Datei `modul_1.zet` ein Fehler aufgetreten ist. ZETA kam mit der Syntax einer Anweisung nicht zurecht und konnte daher keinen Code erzeugen. In einem solchen Fall würde die betreffende Zeile des Quelltextes ausgegeben und die fehlerhafte Stelle mit einem Pfeil markiert. (Wenn ZETA in der Fehlermeldung keine Datei angibt, befindet sich der Fehler in der Hauptdatei.)

ZETA benutzt ein Fehlersystem mit drei Stufen:

Fatale Fehler führen zum sofortigen Abbruch des Kompiliervorgangs. Dies passiert nur in schwer wiegenden Fällen, wenn z.B. kein Speicherplatz mehr vorhanden ist oder eine Datei nicht geöffnet werden kann.

Fehler sind weniger gravierend und ZETA fährt mit der Analyse des Quelltexts fort. Es wird aber keine Spieldatei erzeugt, da der Code korrupt wäre. Wenn zu viele Fehler auftreten, bricht ZETA seine Arbeit mit einem fatalen Fehler ab.

Eine **Warnung** zeigt dem Autor lediglich Ungereimtheiten im Quelltext an, die aber formal richtig und vielleicht sogar beabsichtigt sind. Die Spieldatei wird trotzdem generiert. Mit der Option `-w` kann man Warnungen unterdrücken. (Man sollte sie vor der Veröffentlichung eines Spiels aber trotzdem einmal beachten.)

. . . . .

Fehler, die beim Kompilieren auftreten, sind meistens syntaktische Fehler im Quelltext. Grob eingeteilt gibt es folgende Gruppen von Fehlern:

Ein- und Ausgabefehler

Diese Fehler – oft *I/O*-Fehler (von engl. *Input/Output*) genannt – treten auf, wenn ZETA nicht auf eine Datei zugreifen kann:

```
Die Datei nihil.zet kann nicht geöffnet werden.
```

Hier ist sehr wahrscheinlich der Dateiname in einer `#datei`-Anweisung falsch geschrieben. Auf manchen Systemen wird eine Datei auch blockiert, wenn sie momentan benutzt wird. Auf die Spieldatei kann z.B. nicht geschrieben werden, wenn sie gerade in einem Interpreter zum Testen geöffnet ist.

Ein weiterer *I/O*-Fehler betrifft eingebundene Dateien:

```
Die maximale Verschachtelungstiefe (12) ist erreicht.
```

Das bedeutet, dass mehr als zwölf Dateien eingebunden wurden, die ihrerseits wiederum Dateien einbinden. Das rekursive Einbinden von Dateien ist legal, aber in der Tiefe auf zwölf beschränkt. Hier wurde wahrscheinlich ein „Teufelskreis“ von eingebundenen Dateien gebildet, z.B. wenn eine Datei sich selbst einbindet.

Die Anzahl der Dateien, die nacheinander eingebunden werden können, ist unbegrenzt.

Erreichen der Speichergrenzen

Dies kann bei größeren Projekten passieren, wenn der Speicherplatz für bestimmte temporäre Daten, die während des Kompilierens benötigt werden, nicht ausreicht.

Es wird in der Fehlermeldung meist gesagt, welchen Speicherparameter man vergrößern kann, um mehr Platz zu sachaffen.

Manchmal ist auch der verfügbare Speicher erreicht. Dann muss man den wert einiger Speicherparameter verringern, damit andere Speicherblöcke Platz haben. Mit der Option `-c`, dem so genannten Trockenlauf, kann man sich die Speichereinstellungen anzeigen lassen, ohne das Spiel zu kompilieren.

× Im Trockenlauf wird auch gesagt, wieviel Speicher insgesamt zur Verfügung steht. Dies ist aber nur ein Anhaltspunkt zur Auslagung der Speicherparameter. Zu den erwähnten Blöcken muss ZETA noch Speicher für seine interne Organisation allokieren. Dieser Speicher wird dynamisch verwaltet, wieviel benötigt wird hängt davon ab, wie groß das Spiel ist.

Wenn alles nichts hilft, kann man durch Setzen der Option `-t` einige temporäre Daten, wie z.B. die erzeugten Texte, auf Dateien auslagern. Das macht das Kompilieren etwas langsamer, da ständig auf diese Dateien zugegriffen werden muss, verbraucht aber weniger Speicher. Vor allem die Texte benötigen viel Speicherplatz.

Nicht alle Texte werden in den Speicherbereich für Texte geschrieben, etwas wie

```
...
vorher = {
    nehmen, schieben:
        stop "Keine Chance, der Amboss ist
            einfach zu schwer."
}
...
```

wird direkt im *Z*-Code-Bereich abgelegt. Der *Z*-Code-Bereich kann nicht auf temporäre Dateien ausgelagert werden, aber mit der Option `-v` kann man erzwingen, dass solche Texte stets in den Bereich für den statischen Text geschrieben werden, so dass man mit `-vt` den Speicherbedarf weiter minimieren kann.

Syntaktische Fehler

Diese Fehler haben in der Regel die Form

`Hier wird ein` *Symboltyp* `erwartet.`

und sind meistens Tippfehler, wie Klammern oder Anführungszeichen, die nicht geschlossen werden, Codeblöcke, die nicht mit einer geschweiften Klammer beendet werden oder Anweisungen, bei denen das Semikolon am Ende vergessen wurde. Wenn dies passiert, kann ZETA durcheinander kommen und erzeugt unter Umständen viele Folgefehler. Dann sollte man sich die erste Fehlermeldung ansehen, und diesen Fehler korrigieren. Da die Liste der Folgefehler sehr lang sein kann, gibt es die Option `-k`, die jeden Fehler als kritisch betrachtet und beim ersten Fehler abbricht.

Ansonsten dient diese Fehlermeldung dazu, falsche Operationen zu vermeiden. Einige Operatoren verlangen Operanden eines bestimmten Typs. Der Operator `.` muss links ein Objekt, rechts eine Eigenschaft stehen haben. Ausdrücke wie `12.8` sind im Quelltext unzulässig. (Obwohl ZETA alle Datentypen in Zahlen umwandelt.)

Diese Art der Überprüfung ist nicht ganz vollständig, denn wenn anstelle von falschen Datentypen komplexere Ausdrücke oder Aufrufe von Routinen stehen, wird angenommen, dass aus diesem Ausdruck der korrekte Datentyp resultiert. Der Programmierer muss also

sicherstellen, dass z.B. eine Routine ein Objekt zurückgibt, wenn er einen Aufruf anstelle eines Objekts verwendet.

Eine weitere Möglichkeit, die Überprüfung zu umgehen, ist, Variablen zu verwenden. Während eine Zeile wie

```
wenn (15 in 0) ...
```

vom Compiler moniert wird, ist

```
a = 15;
b = 0;
wenn (a in b) ...
```

syntaktisch korrekt und erzeugt Code, wenn auch nicht unbedingt sinnvollen.

Ein Fehler, der nicht unbedingt auf den ersten Blick ersichtlich ist, taucht oft im Zusammenhang mit Bedingungen in runden Klammern auf. Die Zeile

```
while (x < 24) num(a[x++]), "\n";
```

erzeugt den Fehler

```
Hier wird ein Semikolon erwartet.
```

nach der runden Klammer. Der Bezeichner `while` ist kein gültiges ZETA-Schlüsselwort, und er bezeichnet in anderen Sprachen eine Schleife mit `solange`, so dass Programmierer, die an C oder Java gewönht sind, an dieser Anweisung nichts Ungewöhnliches finden. Der erste Teil wird hier aber als Aufruf der (noch nicht bekannten) Routine `while` angesehen, von der eine Anweisung wie `Text` abgetrennt werden müsste.

Die meisten Fehlermeldungen für Syntax-Fehler sollten jedoch selbsterklärend sein.

Nicht definierte Symbole

Der Fehler

```
Dieser Bezeichner ist unbekannt.
```

ist in der Regel leicht zu beseitigen. Entweder wurde der Bezeichner falsch geschrieben oder er wurde in einem Ausdruck benutzt, bevor er deklariert wurde, so dass ZETA nicht weiß, welchen Datentyp er bezeichnet. Eine Fehlermeldung wie

```
Das Objekt maxAnzahl wurde nicht definiert.
```

kann dagegen etwas irreführend sein. Der Grund dafür ist, dass alle Bezeichner deklariert werden müssen, bevor sie benutzt werden. Die Ausnahme sind Routinen und Objekte, um die Reihenfolge im Quelltext nicht zwingend festzulegen oder um Kreisreferenzen erzeugen zu können.

Der Aufruf einer Routine kann also erfolgen, bevor die Routine im Quelltext definiert wurde. ZETA merkt sich dann den Namen, weiß, dass es sich um eine Routine handelt und erwartet, dass diese Routine später im Quelltext definiert wird. Ist dies nicht der Fall, gibt es einen Fehler, da irgendwo im Code der Aufruf einer unbekannten Funktion steht. Entweder wurde vergessen, die Routine zu definieren (oder die passende Datei einzubinden, in der diese Routine steht), oder aber der Bezeichner wurde falsch geschrieben.

Die Syntax für einen Funktionsaufruf ist durch die runden Klammern, die auch dann angegeben werden müssen, wenn keine Argumente übergeben werden, eindeutig. Bei Objekten sieht dies etwas anders aus. Wenn ZETA einen Bezeichner findet, den es nicht kennt, so vermutet es, dass es ein Objekt ist, wenn der Bezeichner (*a*) nach dem Schlüsselwort `in` in einer Bedingung oder Objektdefinition steht, (*b*) vor einem der Operatoren `.`, `.&` oder `.#`, (*c*) ein allein stehender Bezeichner in einem Funktionsaufruf ist oder (*d*) als Wert in einem Feld oder einer Objekteigenschaft angegeben wurde.

Diese Vorgehensweise erlaubt einige Flexibilität, kann aber auch zu falschen Fehlermeldungen führen. Wenn das bemängelte Objekt nirgends im Quelltext auftauchen soll, handelt es sich wahrscheinlich um einen Tippfehler im Zusammenhang mit einer der oben genannten Annahmen, der schnell behoben sein dürfte.

5.2 Logische Fehler im Programm

Wesentlich schwieriger zu beheben sind so genannte logische Fehler, bei denen legaler ZETA-Code nicht das macht, was er soll und den Interpreter in schwerwiegenden Fällen sogar zum Absturz bringen kann. ZETA hat keinen strikten Kompiliermodus, der Laufzeitfehler abfängt und eine geeignete Fehlermeldung ausgibt, daher ist beim Programmieren einige Sorgfalt nötig, um später schwer zu analysierende Fehler zu vermeiden.

Nachfolgend werden einige Stolpersteine beim Programmieren mit ZETA aufgeführt.

Routinen

Ein häufig gemachter Fehler bei Routinen, die keine Argumente haben, ist, dass man die leeren Klammern weglässt. ZETA weiß aber nur, dass es die Routine aufrufen soll, wenn nach dem Namen der Routine Klammern stehen. Eine Anweisung wie

```
Taste;
```

erzeugt den Fehler „Diese Anweisung bewirkt nichts“, da für ZETA `Taste` nur ein Wert ist, mit dem hier nichts gemacht wird, nämlich die Adresse der Routine. Gefährlich sind falsche Aufrufe mit Zuweisung:

```
x = Taste;
wenn (x == 'j') "Ja!";
```

Hier wird nicht gewartet, bis der Spieler eine Taste drückt, da `Taste` nie aufgerufen wurde. Stattdessen bekommt `x` den Wert einer Adresse im *Z*-Code.

Auch, wenn diese Syntax gefährlich ist, so hat sie doch ihren Sinn. Man könnte z.B. der Variablen `x` den Wert einer Funktion zuweisen und diese dann später über die Variable aufrufen:

```
x = Taste;
...
y = x();
```

Auf diese Weise können Routinen auch an andere Routinen übergeben werden.

Felder

Bei Feldern muss man aufpassen, dass man nicht versucht, außerhalb des definierten Bereichs zu lesen oder zu schreiben. Ein Feld der Dimension `n` hat Einträge mit den Indizes `0` bis `n-1`. Die Einträge fangen nicht, wie man versucht ist anzunehmen, bei eins an. Das folgende Programm

```
Feld eins[10]
Feld zwei = 1, 2, 3, 4, 5;

Hauptprogramm {
    lokal i;

    schleife (i=1; i<=10; i++) eins[i] = 2*i;
    num(zwei[0]);
```

```
}
```

gibt „20“ aus, nicht „1“. Beim Deklarieren von Feldern wird im dynamischen Speicherbereich des *Z*-Codes genau so viel Platz freigehalten, wie für `n` Einträge benötigt wird. Der Speicherplatz von `zwei` schließt direkt an den von `eins` belegten an, so dass der (nicht existierende) Eintrag `eins[10]` auf den Speicherplatz zugreift, der `zwei[0]` zugedacht ist.

× Der Bezeichner eines Feldes ist für ZETA nichts anderes als die Adresse im Speicher, an der der für das Feld reservierte Speicherplatz beginnt, so dass `eins[10]` die beiden Bytes, die an der Adresse `eins+2*10` liest und sie als 16-Bit-Zahl interpretiert. Bei illegalen Indizes greift man mit dieser Methode auf beliebige Daten zu, beim Schreiben auf Einträge mit illegalen Indizes kann man wichtige Daten zerstören.

. . . . .

Eine weitere Schwierigkeit bei Feldern ist die Unterscheidung zwischen Strings und *Word*-Feldern. Eine Deklaration ohne weitere Angaben erzeugt ein *Word*-Feld, dessen Einträge zwei Bytes lang sind und als 16-Bit-Zahl interpretiert werden. Strings werden oft benutzt, um Zeichenketten abzulegen, bei denen es ausreicht, wenn die Einträge die Werte von null bis 255 darstellen können.

Mit eckigen Klammern greift man immer so auf ein Feld zu, wie es seinem Typ entspricht, und Einsteiger sollten immer diese Variante benutzen. Trotzdem kann es Probleme geben, zum Beispiel:

```
String Passwort = "Mississippi";

Routine schreibeText(a) {
    lokal i;
    solange (a[i]) char(a[i++]);
}
```

Ein Aufruf der Routine `schreibeText(Passwort)` schreibt nun „issip“, gefolgt von vielen unerklärlichen Zeichen. Warum? `a` ist kein Feldbezeichner, sondern eine Variable, und daher weiß ZETA nicht, um welchen Typ von Feld es sich handelt. Es nimmt immer an, dass es ein *Word*-Feld ist, es sei denn, das Feld hat einen Feldbezeichner, dem ZETA klar einen String zuordnen kann.

Bei lokalen Variablen, die Felder beschreiben, kann man einen Typ, `String` oder `Feld` mitgeben, so dass ZETA weiß, auf welche Art auf dieses Feld zugegriffen werden soll. Die Routine zum Ausgeben von Textstrings sähe also so aus:

```
Routine schreibeText(String a) {
    lokal i;
```

```
    solange (a[i]) char(a[i++]);
}
```

Mit den Operatoren `->` und `-->` greift man auf Felder zu, als ob sie Strings oder *Word*-Felder wären, ungeachtet dessen, welchen Typ sie tatsächlich haben. Auch hier muss man wieder auf die Dimensionen des Felds aufpassen:

```
Feld w[16];
String b[32];
```

Diese Felder belegen beide 32 Bytes. Je nachdem, mit welchem Operator man diese Felder liest, haben sie Einträge von `->0` bis `->31` oder `-->0` bis `-->15`.

Man kann auch „Mischfelder“ erzeugen, indem man mit verschiedenen Operatoren auf sie zugreift. Die *Z*-Maschine macht dies z.B. beim Aufteilen von Eingabezeilen in Wörter. Diese Methode ist aber sehr unübersichtlich und wird besser vermieden. Besser ist es, mehere Felder zu definieren, so dass gleiche Indizes immer auch zusammengehörige Daten referenzieren.

Felder können, wie oben gesehen, an Routinen übergeben werden. Während Variablen nach Wert übergeben werden (*call by value*), werden Datenstrukturen wie Felder oder Objekte nach ihrer Adresse übergeben (*call by reference*). Das heißt, dass eine Variable stets eine Kopie der Variable für die jeweilige Routine ist, die beim Aufruf angefertigt wird. Innerhalb der Routine kann sie verändert werden, aber dies hat für die aufrufende Routine keine Konsequenzen. Einträge von Feldern, die übergeben werden, bleiben auch nach Verlassen der Routine verändert. Es wird keine lokale Kopie des Felds angelegt, lediglich eine Kopie der Feldadresse. (Diese könnte dann wieder ohne Konsequenzen für andere Routinen geändert werden.)

Folgendes Programm ersetzt daher zwar alle „i“ durch andere Vokale, zerstört aber die einzige im Speicher vorhandene Kopie des Passworts:

```
Routine schreibeText(String a) {
    lokal i;
    solange (a[i]) {
        wenn (a[i]=='i')
            a[i] = Zufall('a', 'e', 'i', 'o', 'u');
        char(a[i++]);
    }
}
```

Folgende Routine schreibt eine veränderte Version des Passworts zerstörungsfrei:

```
Routine schreibeText(String a) {
    lokal i;
    solange (a[i]) {
        wenn (a[i]=='i')
```

```
            char(Zufall('a', 'e', 'i', 'o', 'u'));
        sonst char(a[i]);
        i++;
    }
}
```

Teil III:

Die Adventure-Bibliothek

6 Einführung in die Bibliothek

6.1 Aller Anfang ...

Die Adventure-Bibliothek, oder kurz *z-Lib* ist das Herzstück eines Textadventures. Sie enthält den Parser, die Regeln zur Ausgabe von Objekt- und Raumbeschreibungen und viele vorgefertigte Konzepte für häufig in Adventures vorkommende Begebenheiten. Diese Bibliothek ist so ausgelegt, dass sie leicht verwendbar ist, gleichzeitig aber problemlos an die Wünsche des Autors angepasst werden kann.

Die Beschreibung dieser Bibliothek teilt sich in vier Kapitel auf. Dieses Kapitel beschreibt die grundlegende Vorgehensweise der Bibliothek. In den folgenden Kapiteln werden das Modell der Spielwelt, die Möglichkeiten zur Beschreibung von Objekten, Räumen und Listen und schließlich die Fuktionsweise des Parsers erläutert.

Als Einstieg soll hier als Beispiel die Anfangsszene eines Klassikers betrachtet werden. (Die so genannten klassischen Spiele sind zwar etwas aus der Mode gekommen, aber selbst avantgardistische Adventures halten sich an die Grundregeln aus der Frühzeit des Genres.)

Der folgende Code ist die erste Version eines mit der z-Lib geschriebenen Adventures:

```
Konstante Story = "Das Große Unterweltreich";
Konstante Copyright = "Ein interaktives Plagiat";

#datei "zlib.zin"

Routine Anfang {
    aRaum = Westlich_des_Hauses;
}

Raum Westlich_des_Hauses {
    Name = "Westlich des Hauses";
    Beschreibung = "Du stehst auf freiem Feld, westlich
        eines kleinen weißen Hauses, dessen Tür mit
        Brettern zugenagelt ist. ";
}
```

Dies ist ein minimales Spiel. Es enthält nur einen Anfangstext, einen Raum und keine weiteren Objekte. Trotzdem kann man es kompilieren und sogar spielen. Dadurch, dass die Bibliothek, die sich in der Datei `"zlib.zin"` befindet, eingebunden wird, kennt das Spiel die gängigen Befehle, auch wenn die meisten durch die spärlichen Voraussetzungen eher nutzlos sind.

Die beiden Konstanten `Story` und `Copyright` müssen immer definiert werden, bevor die Bibliothek eingebunden wird. Sie enthalten den Titel des Spiels und die Information über Autor und Copyright des Werks. Diese Angaben werden im Spielbanner zu Beginn des Spiels angezeigt, aber auch wenn der Spieler „Version“ eingibt. Der Hauptteil des Spiels wird allerdings nach dem Einbinden der Bibliothek definiert.

Ein `Raum` ist eine Objektklasse, die in der Bibliothek vordefiniert ist. Jedes Spiel muss mindestens einen Raum enthalten, und jeder Raum muss `Name` und `Beschreibung` haben. Die Routine `Anfang` ist ebenfalls verpflichtend für ein Spiel, in ihr kann ein Anfangstext geschrieben werden und der Spieler muss an seinen Anfangsort gebracht werden.

× Die Bibliothek enthält auch das Hauptprogramm, so dass ein Adventure, das diese Bibliothek nutzt, kein Hauptprogramm definieren muss.

× Die Bibliothek besteht nicht nur aus einer, sondern aus etwa zehn Dateien, die verschiedene Bereiche der Bibliothek enthalten. Diese Dateien werden aber alle aus `"zlib.zin"` heraus aufgerufen, so dass der Autor eines Spiels nur diese Datei einbinden muss.

. . . . .

Eben wurde gesagt, dass die Routine `Anfang` den Anfangsort des Spielers festlegen muss. Das geschieht, indem man die Variable `aRaum` setzt oder das Objekt `Spieler` an den Startpunkt verschiebt.

× Die globale Variable `aRaum` bezeichnet den momentanen Aufenthaltsort des Spielers, ein Objekt der Klasse `Raum`. `Spieler` ist eine Variable, die auf den Objekt zeigt, durch das der Spieler handelt. Wie wir in Abschnitt 7.12 sehen werden, kann sich dieses Objekt ändern.

6.2 Interaktion mit Objekten

Fast alles im Spiel wird durch Objekte dargestellt – die Räume, in denen sich der Spieler aufhält, Gegenstände, die der Spieler mitnehmen kann oder nicht, andere Personen im Spiel und sogar der Spieler selbst.

Im Quelltext unseres Plagiats taucht ein einziges Objekt auf, der Raum mit dem Namen „Westlich des Hauses“. Das gesamte Spiel hat bereits 21 Objekte, zwanzig werden in der Bibliothek definiert, wie zum Beispiel die Himmelsrichtungen und der Spieler. Aber auch der Parser verwendet Objekte, um Objektlisten und -gruppen darzustellen. (Wen interessiert, welche das sind, kann das Spiel einmal mit den Optionen `-yl` kompilieren und sich die

Log-Datei anschauen.)

Um unser Beispiel etwas interaktiver zu gestalten, fügen wir zwei neue Objekte ein:

```
Objekt -> Briefkasten {
    Name = "klein{e} Briefkasten";
    Genus = m;
    Adj = 'klein';
    Subst = 'briefkasten' m, 'kasten' m;
    ist fest, Behälter, schließbar, geschlossen;
}

Objekt -> -> Flugblatt {
    Name = "Flugblatt";
    Genus = n;
    Subst = 'flugblatt' n, 'blatt' n, 'zettel' m;
}
```

Diese Objekte sind sehr gewöhnliche Objekte, sie haben nur die wichtigsten Merkmale: Der `Name` wird ausgegeben, wenn sich das Spiel auf die Objekte bezieht, in einer Raumbeschreibung zum Beispiel. Damit dieser Name mit einem passenden Artikel und gebeugten Adjektiven ausgegeben werden kann, müssen Objekte einen `Genus` haben, der eine der Konstanten `m` (Maskulinum), `f` (Feminimum), `n` (Neutrum) oder `p` (Plural) ist. Das `{e}` in geschweiften Klammern wird bei der Ausgabe durch die passende Endung ersetzt. Alle Details der Ausgabe von Objekten werden in Abschnitt 8.1 beschrieben.

Dann hat jedes Objekt Vokabular, das der Spieler in seiner Eingabe verwenden kann, Adjektive und Substantive. Die Substantive müssen ebenfalls mit einem Genus definiert werden. Vokabeln, also Wörter, die in der Eingabe und nicht in der Ausgabe werdendet werden, werden in einfache Anführungszeichen gesetzt.

Außerdem hat jedes Objekt seinen Platz im Spiel, in ZETA ist das die Lage im Objektbaum. Die anfängliche Position wird in der Kopfzeile der Objektdefinition entweder mit Pfeilen wie hier oder durch Angaben wie `in Briefkasten` angegeben wird. Hier ist der Briefkasten das Tochterobjekt des Raums und das Flugblatt die Tochter des Briefkastens: Der Briefkasten steht direkt im Raum, das Flugblatt ist darin.

Trotzdem sieht man das Flugblatt zu Beginn nicht. Der Briefkasten hat mehrere Attribute, die mit `ist` definiert werden, darunter `geschlossen` und `Behälter`. Das heißt, der Spieler muss den Briefkasten erst öffnen, bevor er an das Flugblatt gelangt. Dass er den Briefkasten einfach öffnen und schließen kann ermöglicht das Attribut `schließbar`. (Mehr über Behälter in Abschnitt 7.3.) Schließlich gibt das Attribut `fest` an, dass man das Objekt nicht einfach mitnehmen kann. Das Flugblatt hat außer Name und Vokaular keine weiteren Attribute, was es im Moment noch zu einem etwas nichts sagenden Objekt macht, wie ein Verlauf unseres Beispiels zeigt:

Das Große Unterweltreich
Ein interaktives Plagiat
Release 1, Seriennr. 050130, z-Lib 0.0 prebeta

Westlich des Hauses
Du stehst auf freiem Feld, westlich eines kleinen weißen Hauses, dessen Tür mit Brettern zugenagelt ist.

Hier ist ein kleiner Briefkasten.

> `untersuche den briefkasten`
Der kleine Briefkasten ist geschlossen.

> `öffne ihn`
Der kleine Briefkasten ist jetzt offen. Im kleinen Briefkasten siehst du ein Flugblatt.

> `nimm das flugblatt`
Du nimmst das Flugblatt aus dem kleinen Briefkasten.

> `lies es`
Auf dem Flugblatt steht nichts.

> `untersuche es`
Du siehst nichts Besonderes an dem Flugblatt.

> `lege das flugblatt in den briefkasten`
Du hast das Flugblatt in den kleinen Briefkasten gelegt.

> `l`

Westlich des Hauses
Du stehst auf freiem Feld, westlich eines kleinen weißen Hauses, dessen Tür mit Brettern zugenagelt ist.

Hier ist ein kleiner Briefkasten.

Im kleinen Briefkasten siehst du ein Flugblatt.

. . . . .

Erweitern wir also die Definition des Flugblatts:

```
Objekt -> -> Flugblatt {
    Name = "Flugblatt";
    Genus = n;
    Adj = 'farbig';
    Subst = 'flugblatt' n, 'blatt' n, 'zettel' m,
        'papier' n;
    Beschreibung = "Ein farbiges Blatt Papier. Auf einer
        Seite steht etwas.";
    Vorher = {
        lesen:
            stop "In großen Lettern steht auf dem Zettel:
                ""Willkommen im Großen Unterweltreich!""

                Darunter steht noch mehr, aber es ist viel
                und klein gedruckt, du hast keine Lust, diese
                Bleiwüste durchzulesen.";
    }
}
```

Es sind zwei Eigenschaften hinzugekommen. Die `Beschreibung` ist bei Objekten der Text, der ausgegeben wird, wenn der Spieler „untersuche den Zettel“ eingibt. Dies könnte in einer Routine geschehen, hier begnügen wir uns mit einem einfachen Text.

Interessanter ist die Eigenschaft `Vorher`. Hier können Aktionen abgefangen werden. Das normale Verhalten der Bibliothek bei „lies Objekt“ ist, einfach „Auf dem Objekt steht nichts“ zu sagen. Das gilt gewiss für die meisten Objekte, für ein Flugblatt ist es eher unangebracht. Bevor dieses Standardverhalten zum Zuge kommt, gibt die Bilbiothek dem Objekt die Chance, ein eigenes Verhalten zu definieren. Es fragt dazu die Eigenschaft `Vorher` des betreffenden Objekts ab, die eine Routine ist. Hier kann zu jedem Befehl etwas programmiert werden. Wenn diese Routine einen wahren Wert zurückgibt, so wird die Aktion des Spielers abgebrochen.

Hier wird nur das Lesen abgefangen. Die Anweisung `stop` gibt den nachfolgenden Text aus, springt aus der Routine und gibt den Wert `wahr` zurück. Damit ist in diesem Fall das Standardverhalten überschrieben worden.

Man kann natürlich auch Text ausgeben und die Aktion weiterlaufen lassen:

```
Vorher = {
    lesen: "Eigenartig: ";
}
```

Nun erfährt der Spieler beim Lesen: „Eigenartig: Auf dem Flugblatt steht nichts.“

× Die Syntax für die Befehle in `Vorher` (und in einigen anderen Eigenschaften, wie wir sehen werden) ist die die Bereichsangabe in `jenach`. (Es ist in der Tat ein implizites `jenach`, bei dem aus der Bibliothek ein Wert vorgegeben wird, hier der momentane Befehl.)

. . . . .

6.3 Aktionen und Befehle

6.4 Erweiterungen der Bibliothek

7 Das Modell der Spielwelt

Für viele häufig vorkommende Begebenheiten in Textadventures hat die Bibliothek bereits vorgefertigte Attribute, Eigenschaften und Befehle. Dieses Kapitel stellt diese Konzepte der Spielwelt vor.

7.1 Räume und Richtungen

Räume sind die Grundelemente der Spielwelt. In und zwischen ihnen kann sich der Spieler in der so genannten Spiellandschaft bewegen. Alles, was sich nicht in einem der Räume befindet, ist dem Spieler nicht zugänglich. Das Konzept der Räume dient also nicht nur dazu, die Spielwelt zu definieren, es schränkt sie also auch ein.

Räume sind Objekte der Klasse `Raum`, die miteinander verbunden sein können. Räume haben kein Mutterobjekt, jeder Raum ist die Wurzel eines Objektbaums.

Hier ist ein solcher Raum:

```
Raum ...
```

Die Räume können untereinander verbunden werden, indem man Richtungen angibt, in die man von einem Raum in den anderen gelangen kann. Üblicherweise werden in Textadventures die Himmelsrichtungen sowie „hoch“, „runter“, „rein“ und „raus“ dazu benutzt, und standardmäßig werden in der Bibliothek folgende Richtungen definiert:

```
Norden          Nordosten          hoch
Osten           Südosten           runter
Süden           Südwesten          rein
Westen          Nordwesten         raus
```

Diese Richtungen sind Objekte der Klasse `Himmelsrichtung`, die Kinder des Hilfsobjekts `Windrose` sind. Wenn der Spieler „gehe nach Norden“ (oder natürlich „n“) sagt, so wird der Befehl `gehen` mit dem Objekt `Norden` ausgeführt. ZETA sendet dann die Nachricht

```
aRaum.Ausgänge(Norden);
```

Wenn die Antwort ein Raum ist, so geht der Spieler dorthin. Ist die Antwort wahr, so passiert nichts, da angenommen wird, dass dem Spieler bereits gesagt wurde, warum. Danach wird die Nachricht

```
aRaum.Ausgänge(nichts);
```

gesendet, um eine individuelle Absage schreiben zu können. Ist die Antwort hier ebenfalls falsch, wird eine Standard-Absage verwendet. Da in der Eigenschaft `Ausgänge` mehrere Fälle abgefragt werden, kann man die Richtungsobjekte wie die Aktionen in `Vorher` usw. mit einem Doppelpunkt den gültigen Blöcken voranstellen:

```
Ausgänge = {
    Norden: antworte Im_Dschungel;
    Süden, Südwesten, Westen:
        stop "Dort geht es steil die Klippe herunter.";
    runter: stop "Besser nicht...";
    Nordwesten: antworte Bei_einer_Ruine;
    nichts: stop "Dort ist der Dschungel zu dicht. Ein
        breiter Pfad führt nach Norden, ein schmalerer
        entlang der Klippe nach Nordwesten.";
}
```

× Natürlich könnte man auch anstatt `nichts` hier `sonst` verwenden. Wenn es aber eine Raumklasse gibt, die in `Ausgänge` eigene Regeln definiert, würden diese mit `sonst` vorzeiteig abgefangen, daher verwendet man hier besser `nichts`.

× Die verwendeten Richtungen sind nicht endgültig. Jeder Autor kann eigene Richtungen als Objekte in der `Windrose` definieren. Diese können sich auf bereits vorhandene Eigenschaften oder auf vom Autor neu definierte beziehen.

× Dem Spieler stehen immer nur die Richtungsobjekte zur Verfügung, die sich im Moment in der `Windrose` befinden. Normalerweise ändert sich der Satz der gültigen Richtungen nicht, man kann aber Richtungen erlauben oder verbieten, indem man die dazugehörigen Objekte in die Windrose hinein oder aus ihr heraus verschiebt.

× Die Definitionen der acht Himmelsrichtungen und der dazugehörigen Eigenschaften werden ignoriert, wenn man vor dem Einbinden der Bibliothek die Konstante `keine_Himmelsrichtungen` definiert.

- **Aufgabe 1**

Definiere für ein Abenteuer zu Lande und zur See die zusätzlichen Richtungen Backbord, Steuerbord, bugwärts und achtern.

. . . . .

Verbindungen zwischen Räumen sind Einbahnstraßen. Nur weil es von einem Raum in eine bestimmte Richtung in einen andere geht, heißt das nicht, dass man aus diesem Raum in die entgegengesetzte Richtung wieder zurück in den Ausgangsraum kommt. Dies muss explizit im zweiten Raum definiert werden.

× ZETA weiß nicht, welche Richtungen sich gegenüberliegen, die Richtungsobjekte haben eine Richtung, aber keine Gegenrichtung.

Außerdem kann man so „verbogene“ Verbindungen herstellen, die nicht nur für die berüchtigten Labyrinthe verwenden kann, sondern auch zum Auflockern der Landkarte:

```
Ausgänge = {
    Südwesten:
        "Du gehst durch das Rosenspalier und folgst
            einem langen Gang, der einen Bogen nach
            Westen schlägt.\p";
        antworte Gartenanlage;
}
```

- **Aufgabe 2**

In Inform werden die Ausgänge aus einem Raum durch Objekteigenschaften beschrieben, so dass man nicht `aRaum.Ausgänge(Norden)`, sondern `aRaum.nach_N()` aufruft. Implementiere das.

- **Aufgabe 3**

Die Spiele von *Magnetic Scrolls* geben eine Liste der möglichen Ausgänge an in der Form: „Du kannst nicht nach Norden gehen, nur nach Osten und Südwesten.“ Implementiere das.

- **Aufgabe 4**

Definiere die Richtung zurück, so dass man mit „gehe zurück“ wieder in den Ausgangsraum gelangt.

7.2 Türen und Verbindungen

Mit dem System der Richtungen kann man Räume direkt miteinander verbinden. Ein weiteres Konzept erlaubt es, indirekte Verbindungen zwischen Räumen zu schaffen: Objekte, die das Attribut `Verbindung` besitzen, können `betreten` werden. Die Eigenschaft `Zielraum` gibt an, was beim Betreten der Verbindung passiert. Genau wie bei den Richtungen von Räumen kann die Antwort hier `wahr` (Das Betreten scheitert, und es wurde gesagt warum), `falsch` („Das Objekt führt nirgendwohin“) oder ein Objekt sein – der Raum, der durch diese Verbindung betreten wird.

Das Betreten einer Verbindung scheitert, wenn diese `geschlossen` ist. Daher kann man Türen gut als `Verbindung` implementieren.

Außerdem können die Richtungen eines Raums anstatt auf einen anderen Raum auch auf eine Verbindung zeigen, die dann wiederum mit ihrem `Zielraum` auf einen anderen Raum

verweist:

```
Raum = Besenkammer {
    Name = "Besenkammer";
    Subst = 'kammer' f, 'besenkammer' f;
    Ausgänge = {
        Osten, raus: antworte Kammertür;
    }
    kein_Weg = "Der einzige Ausgang ist im Osten.";
    Beschreibung = "Diese Besenkammer hat keine
        besonderen Merkmale.";
}

Objekt Kammertür {
    Name = "dunkle Tür";
    Genus = f;
    Subst = 'tür' f, 'kammertür' f;
    Immer = {
        wenn (selbst ist geschlossen)
            "Eine geschlossene Tür";
        sonst "Eine Tür, die offen steht,";
        " führt im Osten heraus.";
    }
    Zielraum = Flur;
    ist Verbindung, geschlossen, schließbar;
}
```

In diesem Fall wird `Ausf(gehen, Osten)` umgewandelt in `Ausf(betreten, Kammertür)`. Wenn man Sonderregelungen zum Verlassen der Kammer nach Osten schreiben will, sollte man also das Betreten der Tür abfangen, um beide Fälle, „gehe nach Osten“ und „gehe durch Tür“, zu berücksichtigen

× Genau wie die Richtungsangaben zu Räumen sind Verbindungen Einbahnstraßen.

× Eine Verbindung kann natürlich alles mögliche sein, eine Brücke etwa, oder ein Pfad. Verbindungen können auch dazu verwendet werden, generelle Konzepte der Verbindung wie zum Beispiel Nachbarräume zu beschreiben.

- **Aufgabe 5**

Wie kann man eine Tür erzeugen, die auf beiden Seiten, also in beiden Räumen ist?

- **Aufgabe 6**

Wie könnte man die Himmelsrichtungen in einem Spiel komplett abschaffen?

. . . . .

Objekte, die das Attribut `schließbar` besitzen, kann man `öffnen` und `schließen`. Eine Tür würde man also `schließbar` machen, wie im Beispiel oben.

Der Zustand des Objekts wird durch das Attribut `geschlossen` beschrieben, das entweder gesetzt oder gelöscht ist. Es gibt kein Attribut „offen“. Da der Normalzustand offen, genauer gesagt nicht `geschlossen` ist, kann man normalerweise durch eine Verbindung einfach hindurchgehen.

× Der Spieler kann `Verbindungen` nicht mitnehmen, sie werden als fest betrachtet.

- **Aufgabe 7**

Programmiere eine Planke, die der Spieler mitnehmen und als Verbindung zwischen zwei Seiten einer Schlucht benutzen kann.

. . . . .

Manche Objekte, in Adventures sind das typischerweise Türen, sind `abschließbar`, man kann sie `aufschließen` und `abschließen`, wenn sie `geschlossen` sind. Abgeschlossene Objekte haben dann zusätzlich das Attribut `abgeschlossen`, sie lassen sich nicht öffnen.

Der passende Schlüssel zur Tür ist dann ein Objekt, dessen Eigenschaft `Schlüssel_für` die Tür enthält.

× `Schlüssel_für` kann auch eine Liste von Objekten enthalten, die mit dem Schlüssel aufgeschlossen werden. Ist `Schlüssel_für` ein Text, so wird das Aufschließen mit diesem Text abgebrochen, der Schlüssel passt auf kein Schloss. Es kann auch eine Routine angegeben werden, die das aufzuschließende Objekt als Argument nimmt. Eine wahre Antwort lässt das Aufschließen zu, eine falsche Antwort unterbindet es.

× Auch Schlüssel, die auf kein Objekt im Spiel passen, sollten die Eigenschaft `Schlüssel_für` definieren, da sie ein Objekt als Schlüssel kennzeichnet. Dann enthält diese Eigenschaft den Wert `nichts`.

× Ein Objekt, das `abgeschlossen` aber nicht `abschließbar` ist, kann nicht mit einem Schlüssel aufgeschlossen werden. Andere Mechanismen, etwa Codekarten oder Fernverriegelungen, müssen dann extra programmiert werden.

- **Aufgabe 8**

Zum Testen eines Spiels mit vielen Türen kann ein Generalschlüssel, der auf jede Tür passt, nützlich sein. Erzeuge einen.

7.3 Behälter, Ablagen und Unterobjekte

Die meisten Objekte liegen nicht einfach auf dem Boden herum, sondern befinden sich auf anderen Objekten: Auf Tischen, in Regalen, in Schränken und Kisten. Wenn ein Objekt eine `Ablage` ist, kann man andere Objekte darauf ablegen, und in Objekte mit dem Attribut `Behälter` können andere Objekte hineingelegt werden. Ein Objekt kann nicht `Ablage` und `Behälter` gleichzeitig sein.

Die Befehle für `Behälter` sind `hereinlegen` und `herausmehmen`, bei einer `Ablage` lauten diese Befehle `darauflegen` und `herunternehmen`. Zu jedem dieser Befehle kann der Behälter oder die Ablage als `aObj2` reagieren.

× Jede Ablage und jeder Behälter können nur eine beschränkte Anzahl von Objekte aufnehmen, die in ihrer Eigenschaft `Kapazität` festgelegt ist. Diese ist per Default 255, was in der Praxis Platz für beliebig viele Objekte bietet. Die Kapazität kann natürlich auch eine Routine sein.

In Raumbeschreibungen, im Inventar des Spieler oder beim Untersuchen wird der Inhalt von Behältern und Ablagen angezeigt. Es wird ein Satz wie

> Auf dem Bücherbord siehst du eine Streichholzschachtel, einen kleinen Drachenbaum und einen Almanach.

ausgegeben. Diesen generischen Satz kann man aber für jedes Behälter- oder Ablagenobjekt anpassen:

```
Inhalt = {
    "Du siehst ", einen(Liste), " auf dem Bücherbord."
}
```

Das Hilfsobjekt `Liste` repräsentiert die passenden Objekte in Listenform. Mit `ein` wird diese Liste ausgegeben, mehr dazu in Abschnitt 8.1.

× Es gibt auch eine Eigenschaft `kein_Inhalt`, die ausgegeben wird, wenn der Behälter oder die Ablage keine Tochterobjekte haben. Dann würde „Das Bücherbord ist leer“ auch in Raumbeschreibungen ausgegeben.

× Auch Räume können die Eigenschaft `Inhalt` haben, um den Standardsatz „Hier ist ein …“ zu ersetzen. Wie eine Raumbeschreinung genau aussieht, wird in [nicht gefunden] beschrieben.

× Ein wahrer Wert der globalen Variable `außerdem` bedeutet, dass dieser Liste einzelne Objekte vorangestellt wurde, weil sie eine `Erst`- oder `Immer`-Beschreibung haben. Dies

kann man zum Anpassen der Inhaltsangabe verwenden.

- Mann kann auch selbst den Inhalt eines Objekts ausgeben mit

```
zeige_Inhalt(obj, Umbruch)
```

Diese Routine zeiht den Inhalt des Objekts `obj`. Diese Anzeige ist rekursiv, so dass auch der Inhalt von Behältern oder Ablagen, die in oder auf `obj` sind, aufgelistet werden. Diese Routine berücksichtigt `Erst` und `Immer`, das heißt die Liste wird ausgegeben, als ob sie in einer Raumbeschreibung auftaucht.

Wenn `Umbruch` gleich `wahr` ist, wird vor der Inhaltsliste ein Absatz geschrieben, bei anderen wahren Werten von `Umbruch` beim ersten Mal ein Zeilenumbruch, vor eventuell folgenden Listen ein Absatz geschrieben.

- Die Präpositionen, die verwendet werden, sind „in“ und „aus“ bei einem `Behälter`, „auf“ und „von“ bei einer `Ablage`. Man kann also Objekte, die eigentlich eine Ablage sind, als Behälter definieren, wenn man die Präposition richtig ausgeben möchte, zum Beispiel „im Regal“. Die Bibliothek erkennt, wenn ein Objekt in eine Ablage oder auf einen Behälter gelegt werden soll und lenkt den Befehl passend um.

. . . . .

Objekte auf einer `Ablage` sind immer sichtbar. `Behälter` verhalten sich hier etwas anders: sie können `geschlossen` sein, dann sind die Objekte darin nicht sichtbar.

Genau wie bei den Verbindungen werden Behälter, die nicht `geschlossen` sind, als offen betrachtet. Um sie `öffnen` und `schließen` zu können, muss man sie `schließbar` machen und natürlich können Behälter auch `abschließbar` sein.

Behälter können auch das Attribut `transparent` besitzen, dann kann man ihren Inhalt immer sehen. Wenn der Behälter `geschlossen` ist, sind die Objekte darin allerdings für den Spieler nicht erreichbar. Das Attribut `transparent` hat für Ablagen keine Bedeutung.

. . . . .

Das Konzept der Behälter und Ablagen eignet sich vor allem zum Verstauen beweglicher Objekte, der Inhalt der Ablagen und Behälter ändert sich in der Regel. Ein Objekt kann aber auch Unterobjekte haben, die getrennt vom Hauptobjekt betrachtet werden, die aber fest mit diesem verbunden sind, wie zum Beispiel Knöpfe an einem Radio oder Anker und Segel eines Modellschiffs.

Die Unterobjekte eines Objekts werden einfach in dessen Eigenschaft `Unterobjekte` als Liste angegeben. Sie befinden sich außerhalb des Objektbaums und haben kein Mutterobjekt. Immer, wenn ein Objekt sichtbar ist, sind auch dessen Unterobjekte sichtbar.

× Auf diese Weise kann man auch Objekte definieren, die gleichzeitig Behälter und Ablagen sind. Hier zum Beispiel eine Waschmaschine. Die Wäsche kommt in die Trommel, andere Gegenstände können oben auf die Maschine gelegt werden.

```
Objekt -> Waschmaschine {
    Name = "Waschmaschine";
    Genus = f;
    Subst = 'maschine' f, 'waschmaschine' f,
        'trommel' f, 'waschtrommel' f, 'deckel' m;
    Inhalt = {
        "In der Trommel der Waschmaschine siehst du ",
        ein(Liste), ".";
    }
    Unterobjekte = WM_Ablage;
    umleiten = {
        darauflegen2, herunternehmen2:
            antworte WM_Ablage;
    }
    ist Behälter, fest, nicht geschlossen, schließbar;
}

Objekt WM_Ablage {
    Name = "Deckel der Waschmaschine";
    Genus = m;
    Inhalt ={
        "Oben auf der Waschmaschine lieg",
            v_t(Liste), _, ein(Liste), ".";
    }
    ist Ablage, verborgen;
}
```

Die beiden Befehle für Ablagen werden nun umgeleitet auf das Unterobjekt `WM_Ablage`. Da dieses Objekt kein eigenes Vokabular hat, kann es nie direkt angesprochen werden und agiert hier nur als Ablage. Inhalte von Unterobjekten werden automatisch angezeigt, so dass hier kein zusätzlicher Code nötig ist. (Mit der Eigenschaft `umleiten` wird das angesprochene Objekt vor der Überprüfung der Gültigkeit umgelenkt, so dass auch „nimm alles von der Waschmaschine“ funktioniert. Mehr dazu in Abschnitt 9.6.)

× `Unterobjekte` kann auch eine Routine sein, die ein Objekt als Argument hat. Wenn das Objekt ein Unterobjekt sein soll, muss mit einem wahren Wert geantwortet werden.

!! Die Methode, eine Routine für `Unterobjekte` zu benutzen, ist zwar recht flexibel, aber auch sehr ineffektiv. Es ist vermutlich besser, sich ändernde Unterobjekte in frei

gehaltene Einträge von `Unterobjekt` nachzutragen, oder sie durch Überschreiben mit `nichts` wieder zu entfernen.

7.4 Begehbare Objekte, Sitze und Fahrzeuge

Nicht immer befindet sich der Spieler direkt in einem Raum, viele Spiele implementieren Objekte, die man betreten kann. Beispiele für solche Objekte, die nicht groß genug sind, um ein eigener Raum zu sein, sind Podeste, mannshohe Käfige, Fahrzeuge oder abgetrennte Bereiche eines Zimmers wie Kochnischen und begehbare Kleiderschränke.

Solche Objekte bekommen das Attribut`betretbar`, sie können dann `betreten` oder `verlassen` werden. In der Regel sind betretbare Objekte außerdem ein `Behälter` oder eine `Ablage`. Als Beispiel

```
Objekt -> Schrank {
    Name = "groß{e} Kleiderschrank";
    Genus = m;
    Adj = 'groß', 'enorm';
    Subst = 'schrank' m, 'kleiderschrank' m;
    Beschreibung = "In diesem Kleiderschrank
        könnte man eine genze Famile verstecken.";
    Immer = "Ein enorm großer Kleiderschrank
        steht an der Wand.";
    Innenansicht = {
        "Du stehst in einem großen Kleiderschrank,
        zwischen den Kleidern deiner Frau.";
        wenn (selbst ist geschlossen)
            " Der Schrank ist geschlossen.";
    }
    ist betretbar, Behälter, fest;
}
```

`Innenansicht` ist eine Eigenschaft, die ausgegeben wird, wenn sich der Spieler in diesem Objekt befindet. Wenn der Raum, in dem dieses Objekt steht, noch sichtbar ist, weil das Objekt nicht `geschlossen` ist, oder weil das Objekt `transparent` oder eine `Ablage` ist, wird die `Innenansicht` an die Raumbeschreibung angehängt, wenn das Objekt die Sichtgrenze ist, wird nur die Innenansicht gezeigt:

> *Im großen Kleiderschrank*
> Du stehst in einem großen Kleiderschrank, zwischen den Kleidern deiner Frau.
> Der Schrank ist geschlossen.
>
> Giovanni, der Nachbar, ist hier. Er friert.

Außerdem siehst du hier eine Bauanleitung und eine kleine Schachtel.

× Oft ist es einfacher, ein betretbares Objekt als einen anderen Raum zu implementieren, esonders dann, wenn man die Objekte im Raum aus diesem Objekt heraus nicht sehen kann.

. . . . .

Objekte, auf denen man sitzen oder liegen kann, sind auch `betretbar`. ZETA unterscheidet zwischen den Befehlen `sitzen`, `stehen` und `liegen`, führt sie aber alle auf `betreten` zurück.

Wenn es wichtig ist, in welcher Position man auf einem Objekt ist – zum Beispiel bei einem Sofa, auf dem man liegen, sitzen oder, seltener, stehen kann – muss dieses Objekt die Eigenschaft Sitzpositionen besitzen. Diese Eigenschaft enthält eine Liste der erlaubten Positionen, die erste Position ist die, die verwendet wird, wenn die Position aus dem Befehl nicht eindeutig hervorgeht. Ein Sofa, auf dem man sitzen und liegen kann, sähe so aus:

```
Objekt -> Sofa {
    Name = "rot{e} Ledersofa";
    Genus = n;
    Adj = 'rot', 'teuer';
    Subst = 'sofa' n, 'ledersofa' n, 'couch' f;
    Beschreibung = "Sehr extravagant und sehr teuer.";
    Sitzpositionen = liegen, sitzen;
    Immer = "Ein rotes Ledersofa steht unter dem
        Fenster.";
    ist fest, betretbar, Ablage;
}
```

Wenn der Spieler sagt „gehe aufs Sofa“ wird dies als der erste Eintrag von `Sitzpositionen`, also als liegen, interpretiert. Man kann auf dem Sofa nicht stehen. Die momentan eingenommene Sitzposition wird in der globalen Variable `Sitzposition` abgelegt.

× Mit den Routinen `v_sitz()`, `v_setz()` und `v_sitzt()` sowie `in_oder_auf(obj)` kann man die `Innenansicht` anpassen, wenn man will. Die standardmäßig verwendete Beschreibung ist:

```
"Du ", v_sitzt(), _, in_oder_auf(x1), _, einem(x1), ".";
```

. . . . .

Ein Fahrzeug ist ein Objekt, das sich mit dem Spieler von einem Ort zum anderen bewegt, wenn sich der Spieler darin befindet. Fahrzeuge bekommen neben dem Attribut `betretbar`

das Attribut `Fahrzeug`:

```
Objekt -> Skateboard {
    Name = "Skateboard";
    Genus = n;
    Adj = 'grell', 'bunt', 'grellbunt';
    Subst 'skateboard' n, 'board' n;
    Beschreibung = "Das Skateboard hast du von deinem
        Bruder im Austausch für 150 Plastiksoldaten
        der Roten Armee bekommen. Die einst grellen
        Farben sind mittlerweile durch steten Gebrauch
        etwas ertäglicher geworden, in einer gezackten
        Schrift steht auf dem Board ""Skate or Die!""";
    Immer = "Dein Skateboard steht hier, bereit, dich
        in einem wilden Teufelsritt durch das Haus
        zu befördern.";
    Sitzpositionen = stehen, sitzen;
    Vorher = {
        fahren: wenn (Sitzposition != stehen)
            stop "Dazu musst du dich auf das Board
                stellen.";
        lesen: stop """Skate or Die!""" Yeah!;
    }
    ist Fahrzeug, betretbar, Ablage;
}
```

Wenn der Spieler nun auf dem Skateboard steht und die normalen Befehle zur Fortbewegung eingibt, wie „raus“ oder „gehe auf die Terrasse“, wird der Befehl `fahren` ausgeführt, `aObj` ist dabei das Skateboard, `aObj2` das Richtungs- oder Verbindungsobjekt. Die Fortbewegung erfolgt genau so, als ob der Spieler zu Fuß unterwegs wäre – Besonderheiten müssen extra programmiert werden. Wenn das Fahren erfolgreich ist, befindet sich der Spieler immer noch auf dem Skateboard, und dies befindet sich im Zielraum.

Manche Objekte sind zu schwer, um sie aufzuheben, sie können aber herumgeschoben werden. Auch dieses Verhalten wird durch das Attribut `Fahrzeug` beschrieben: Ein Objekt, das nicht `fest` und ein `Fahrzeug` ist, kann mit dem Befehl `verschieben_nach` in einen anderen Raum verschoben werden, genau wie bei `fahren` sind die Objekte hier das zu verschiebende Fahrzeug und das Ziel, eine typische Eingabe wäre „rolle die Schneekugel nach Osten“. Hier eine Schubkarre:

```
Objekt -> Schubkarre {
    Name = "Schubkarre";
    Genus = f;
    Subst = 'schubkarre' f, 'schubkarren' m, 'karre' f,
        'karren' m;
    Vorher = {
        nehmen: stop "Die Schubkarre ist zu schwer, um
            sie mit dir herumzuschleppen, du kannst sie
```

```
            aber von einem Ort zum anderen schieben.";
    }
    Meldung = {
        verschieben_nach: "\pUff! Du setzt die
            Schubkarre wieder ab.";
    }
    Erst = "Eine Schubkarre steht hier.";
    ist Behälter, Fahrzeug;
}
```

Die Meldung „Uff! …“ wird ausgegeben, nachdem die Karre erfolgreich verschoben wurde. Man kann in `Vorher` natürlich auch eine Meldung vor der neuen Raumbeschreibung ausgeben, aber man weiß ja nicht immer, ob das Verschieben gelingt. Die Karre wird beim Betreten des neuen Raums nicht in der Beschreibung erwähnt, nachher schon.

× Das Fahren und das Verschieben funktioniert nur, wenn sich das Fahrzeug direkt im Raum befindet. Wenn also das Skateboard auf einem Podest steht, kann der Spieler nicht damit fahren.

× Wenn ein `Fahrzeug` nicht `fest` ist, kann man es immer verschieben. Im Beispiel des Skateboards ist das aber nicht besonders sinnvoll, deshalb sollte man das Verschieben hier mit einer geeigneten Meldung abfangen, genauso wie das Aufheben bei der Schubkarre explizit abgefangen wird.

× Normalerweise ist bei `gehen` das zweite Objekt null. Wenn `gehen` jedoch von `fahren` oder `verschieben_nach` aufgerufen wird, ist `aObj2` das Fahrzeug. So kann man seinen Code für die Richtungen anpassen:

```
Ausgänge  = {
    Norden:
    wenn (aObj2) stop "Mit ", dem(aObj2), " kommst
        du nicht durch den schmalen Durchlass.";
    antworte Kammer;
}
```

× Der Befehl `fahren` hat auch eigenes Vokabular, so dass man auch „fahre mit dem Skateboard in die Küche“ sagen kann. Das Konzept der Fahrzeuge ist aber weitreichend, so dass man eventuell eigene Verben wie „reite“ oder „rudere“ für den Befehl `fahren` definieren sollte.

- **Aufgabe 9**

Konstruiere ein Ruderboot, so dass man zu Fuß nur auf dem Land, mit dem Boot nur auf dem Wasser fahren kann.

7.5 Kleidung

Objekte, die der Spieler an- und ausziehen kann, bekommen das Attribut `Kleidung`. Mit den Befehlen `anziehen` und `ausziehen` kann der Spieler diese Kleidung dann anlegen. Am Körper getragene Kleidung ist, genau wie die anderen Bsitztümer des Spielers, eine Tochter des Objekts `Spieler`. Um dies zu unterscheiden, bekommen Kleidungsstücke das Attribut `angezogen`, wenn sie getragen werden. Dieses Attribut hat nur eine Bedeutung, wenn dss betreffende Objekt beim Spieler ist und sollte auch nur in diesem Zusammenhang verwendet werden.

× Natürlich laufen die wenigsten Spieler nackt herum – eine Ausnahme ist hier Frank Borgers *Eden*. Es wird vielmehr angenommen, dass der Spieler ihm angemessene Kleidung trägt, und sich das An- und Ausziehen nur auf besondere Kleidungsstücke bezieht wie Raumanzüge, Schutzhelme oder leisetretende schwarze Schuhe mit Kreppsohlen.

7.6 Objekte zum Ein- und Ausschalten

Ein Objekt, das das Attribut `schaltbar` besitzt, lässt sich mit den Befehlen `einschalten` und `ausschalten` in einen der beiden Zustände „ein“ und “aus“ bringen. Diese Zustände werden durch das Setzen oder Löschen des Attributs `eingeschaltet` beschrieben. Ein Objekt ist also ausgeschaltet, wenn es `schaltbar`, aber nicht `eingeschaltet` ist.

Solche Objekte können Radios, Kaffeemaschinen oder Taschenlampen sein. Auch eine Fackel wäre ein schaltbares Objekt, obwohl man vielleicht das Verb „schalten“ hier vermeiden und stattdessen „anzünden“ oder „löschen“ verwenden würde.

Was jedoch genau mit dem angeschalteten Objekt passiert, muss der Autor selbst berücksichtigen:

```
Objekt Taschenlampe {
    Name = "Taschenlampe";
    Genus = f;
    Subst = 'lampe' f, 'taschenlampe' f, 'leuchte' f;
    Nachher = {
        einschalten:
            mache selbst Lichtquelle;
        ausschalten:
            mache selbst nicht Lichtquelle;
    }
    ist schaltbar, nicht eingeschaltet;
}
```

Wenn keine `Beschreibung` angegeben wird, wie hier, wird beim Untersuchen von schaltbaren Objekten einfach gesgt: „Die Taschenlampe ist an/aus.“

× Es gibt auch einen Befehl `schalten`, der ein Objekt immer in den jeweils anderen Zustand schaltet und nach `einschalten` oder `ausschalten` umgeleitet wird.

7.7 Lesen und nachschlagen

Auf manchen Objekten steht etwas geschrieben, das man `lesen` kann. ZETA unterscheidet zwischen `lesen` und `untersuchen`. Alles, was man auf einem Objekt lesen kann, muss explizit in `Vorher` abgefangen werden, sonst „steht nichts“ auf dem Objekt.

Oft ist das, was auf dem Objekt steht auch dessen Hauptmerkmal, wie zum Beispiel bei einem Warnschild. Dann kann man das `untersuchen` und das `lesen` zusammenfassen:

```
Deko -> Warnschild {
    Name = "Warnschild";
    Genus = n;
    Subst = 'schild' n, 'warnschild' n,
        'warnung' f;
    Vorher = {
        lesen, untersuchen:
            stop "Unter einem  Totenkopf mit
            gekreuzten Knochen steht: ""Achtung,
            Alligatoren! Schwimmen strengstens
            verboten!"" Interessant ist das
            Kleingedruckte: ""Überlebende werden
            strafrechtlich verfolgt.""";
    }
}
```

. . . . .

Ein Sonderfall des Lesens ist das `nachschlagen` von Stichwörtern in Lexika oder Fachbüchern. Das Nachschlagewerk ist `aObj`. Die Schwierigkeit hier ist, `aObj2` eindeutig festzulegen, denn es kann ja jedes beliebige Objekt dort stehen. Es muss auch nicht unbedingt anwesend sein und der nachzuschlagende Begriff kann auch auf mehrere Objekte zutreffen.

Zum Ermitteln von Stichwörten gelten folgende Regeln: Zu jedem Stichwort muss ein Objekt existieren, das das Attribut `Stichwort` hat. Dieses Objekt kann ein normales Objekt im Spiel sein, muss es aber nicht.

Aus den Objekten, auf die die Eingabe zutrifft, wird dann nach folgendem Schema ein Treffer ausgewählt: Ein Objekt, das ein `Stichwort` ist, bekommt vier Punkte, momentan sichtbare Objekte zwei und Objekte, die das Attribut `bekannt` haben, bekommen einen. Das Objekt mit der höchsten Punktzahl wird genommen. Wenn es mehrere Objekte mit der selben Bestpunktzahl gibt, so wird das erste stillschweigend ausgewählt. Wenn es das Objekt aus der Angabe nicht gibt, so ist `aObj2` das Objekt `unbekanntes_Objekt`, das in der Bibliothek definiert wird. Das bedeutet, dass in jedem Fall ein Objekt eindeutig gefunden wird und der Parser nie nachfragt.

× Ein Objekt ist `bekannt`, wenn es einmal in einer Raumbeschreibung aufgetaucht ist, oder wenn es der Spieler explizit `bekannt` gemacht hat.

× Die Regel, dass anwesende Objekte stärker gewertet werden, ist für das Befragen von Personen gedacht, bei dem dieselben Regeln angewandt werden. Dies ist ein Sonderfall der `bekannt`-Regel.

× Durch die starke Wertung des Attributs `Stichpunkt` können gezielt Objekte hervorgehoben werden. Wenn es im Spiel eine Münze mit dem Konterfei des Königs gibt und einen „König“, der `Stichpunkt` ist, im Spiel selbst dem Spieler aber nie begegnet, wird trotzdem der König nachgeschlagen.

× Natürlich können einzlne Objekte immer noch nachträglich stärker gewichtet werden, siehe dazu [nicht gefunden].

- **Aufgabe 10**

Definiere einen Sternenatlas, in dem man verschiedene Sterne nachschlagen kann.

7.8 Andere Personen im Spiel

Mit dem Attribut `Person` kann man ein Objekt als Mitmensch oder als anderes vernunftbegabtes Lebewesen definieren. Diese Definition hat einigen Einfluss auf das Verhalten der Bibliothek, aber im begrenztem Umfang auch auf das Verhalten des Parsers.

Eine `Person` ist ein zunächst ein Objekt wie alle anderen auch, sie kann die Eigenschaften `Vorher`, `Nachher` usw. besitzen. Man kann Personen allerdings nicht aufheben, und in Raumbeschreibungen werden sie immer gesondert aufgeführt, entweder mit den Texten `Erst` oder `Immer`, die in diesem Fall dieselbe Bedeutung haben, oder mit dem generischen Text „Ein Holzfäller ist hier.“

Es gibt einige Befehle, die besonders für andere Personen im Spiel gedacht sind: `geben`, `zeigen`, `fragen`, `erzählen`, `angreifen` und sogar `küssen`. Diese Befehle machen

aber nichts, wenn man nichts ausdrücklich in `Vorher` definiert. Die Person lehnt Gaben höflich ab, zeigt sich nicht beeindruckt von Gezeigtem und weiß keine Antwort auf konkrete Fragen. Auf einen Angriff schließlich hat die Bibliothek die belehrende Antwort „Gewalt ist keine Lösung.“

Eine Person kann auch andere Gegenstände bei sich tragen, die dann als Tochterobjekte der Person definiert werden. Wenn die Person das Attribut `transparent` hat, sind diese Objekte sichtbar, sie werden dann bei der Beschreibung aufgezählt. Eine Person kann auch die Eigenschaft `Inhalt` haben, mit der diese Beschreibung individuell angepasst werden kann.

Tochterobjekte einer Person, die nicht `transparent` ist, bewegen sich mit der Person, bleiben dem Spieler aber verborgen.

× Personen und Tiere halten sich nicht unbedingt immer am selben Ort auf. Um sie im Spiel herumwandern zu lassen, sei es entlang einer vorgegebenen Route oder auf mehr oder weniger zufällige Weise, benutzt man am besten einen so genannten Dämon. Wie das geht, steht in Abschnitt 7.13.

7.9 Kommunikation

Zur Kommunikation mit anderen gibt es verschiedene Ansätze. Ein einfaches System ist, andere nach bestimmten Dingen zu `fragen` oder ihnen Sachen zu `erzählen`. Die Stichwörter für `fragen` und `erzählen` werden genauso ermittelt wie beim Nachschlagen:

```
Objekt Polizist
Name = "Polizist";
Genus = m;
Subst = 'polizist' m, 'pom' m, 'polizeiobermeister' m,
    'obermeister' m, 'bulle' m;
Beschreibung = "Dieser Polizist trägt die Kennzeichen
     eines Polizeiobermeisters, drei grüne Sterne und
     einen Oberlippenbart.";
Immer = "Ein Polizist beobachtet gelangweilt das
     eher uninteressante Geschehen hier.";
Vorher = {
    fragen: jenach (aObj2) {
        Wache: """Mein Revier ist die vierte
            Polizeiinspektion in Obersendling""
            proklamiert der Polizist stolz.";
        Polizist: """Ich bin schon seit fünfzehn Jahren
            bei der Bereitschaftspolizei.""";
        sonst: """Darüber darf ich Ihnen leider nichts
            sagen, Dienstgeheimnis."" Der Polzist wirkt
            nicht gerade überzeugt von dieser faulen
```

```
            Ausrede.";
        }
        stop;

        erzählen: jenach (aObj2) {
            Mord: """Kommen Sie mit, das müssen wir
                melden!""" Der Polizist zerrt dich sanft
                aber bestimmt zur nächsten Wache.";
                verschiebe selbst nach Wache;
                gehe_zu(Wache);
            sonst: """Ach ja? Wissen Sie, ich höre jeden
                Tag so viele Geschichten"" Der Polizist
                winkt müde ab.";
        }
        stop;

        küssen: stop "Nicht bei dem Schnäuzer!";
        geben: stop """Nein, das wäre Bestechung!""";
        zeigen: stop """Wenn Sie Beweismaterial oder
            Fundsachen abgeben wollen, gehe Sie doch bitte
            zur Wache. Danke.""";
        angreifen: stop "Der Polizist hat eine Pistole,
            einen Gummiknüppel, Handschellen und ein
            Funkgerät, mit dem er viele ähnlich
            ausgestattete Kollegen herbeirufen kann.
            Besser nicht.";
    }
```

Auch hier ist `aObj2` das Hilfsobjekt `unbekanntes_Objekt`, wenn die Eingabe keinem realen Objekt im Spiel entspricht.

. . . . .

Eine weitere Möglichkeit ist, anderen Personen Anweisungen zu geben. Diese Anweisungen sind genau so aufgebaut wie die Anweisungen des Spielers an den Protagonisten, nur, dass der Empfänger der Anweisungen, der so genannte `Akteur` der Anweisung mit einem nachfolgenden Komma vorangestellt wird: „Kapitän Müller, geben Sie mir die Pistole“.

In diesem Fall ist der `Akteur` Kapitän Müller, der `Befehl` ist `geben`, `aObj` ist der Spieler und `aObj2` die Pistole.

Wenn der Befehl nicht vom Spieler selbst ausgeführt wird, wird die Ausführung weitergeleitet an die Eigenschaft `Anweisungen` des Akteurs:

```
    Anweisungen = {
        gehen: stop """Nein, ich bleibe hier.""";
        nicht_verstanden:
            stop "Der Polizist schaut dich verständnislos an.";
```

```
        sonst: stop """Ich bin nur meinem Dienstherrn
            verpflichtet, sonst niemandem.""";
    }
```

Dieses Beispiel ist nur sehr kurz, eine Anweisung an den Polizisten bewirkt nur eine Absage. Ein solches Verhalten ist wohl der Regelfall. Wenn Akteure komplexere Handlungen ausführen sollen, muss dies explizit implementiert werden. Die Bibliothek hat Regeln dafür, was passiert, wenn der Spieler einen Gegenstand aufhebt, für beliebige Akteure existieren diese Regeln nicht.

Wenn die `Anweisungen` des Akteurs einen falschen Wert zurückgeben, wird der Befehl `ansprechen` mit dem Akteur als `aObj` ausgeführt. Dies bietet die Möglichkeit, eine allgemeine Form der Absage zu implementieren, die Einzelheiten der Anweisung sind dem Autor hier nicht mehr zugänglich, da die Variablen `Befehl`, `aObj` und `aObj2` mit den Werten `Ansprechen`, `Akteur` und `nichts` überschrieben wurden. Der `Akteur` wechselt beim ansprechen wieder auf den Spieler. Wenn der Befehl `ansprechen` nicht abgefangen wird, ist die lapidare Antwort der Bibliothek „Der Kapitän hat Besseres zu tun.“

× `ansprechen` ist ein Pseudo-Befehl, es gibt kein Satzmuster, das ihn erzeugen könnte.

× Der `Akteur` ist natürlich gleich dem `Spieler`, wenn er nicht explizit angegeben wird. „Du, nimm die Axt“ ist also gleichbedeutend mit „nimm die Axt“.

× Formen von „du“ (oder „ihr“ oder „Sie“) beziehen sich immer auf den `Akteur`, Formen von „ich“ auf den `Spieler`. Bei „Schau dich an“ ist `aObj` der Spieler, bei „Annika, schau dich an“ der `Akteur`, Annika.

× Ein Objekt, das die Eigenschaft `Anweisungen` besitzt, muss nicht unbedingt eine Person sein. Jedes beliebige Objekt kann theoretisch angesprochen werden.

× Man kann verschiedenen Personen verschiedene Verben zuordnen, wenn man will. Dies ist sinnvoll für Situationen, in denen Anweisungen nicht an Menschen, sondern an Maschinen gegeben werden („Gamma, Schutzwall ein“) oder wenn für manche Menschen andere Regeln, wie zum Beispiel andere Sprachen, gelten. So können Verben beliebig eingeschränkt oder erweitert werden, auch für den Spieler selbst. Dies geschieht mit der Eigenschaft `Verben`, die in Abschnitt 9.2 erklärt wird.

7.10 Dunkelheit

Manche Spiele, insbesondere die „klassischen“ haben dunkle Räume, in denen man nichts sehen kann solange man keine geeignete Lichtquelle bei sich trägt.

Das Attribut `Lichtquelle` (oder `hell`) markiert ein Objekt als leuchtend. Der Spieler hat genug Licht, um sehen zu können, wenn eine Lichtquelle in Sichtweite ist.

× Abgesehen von Sonderfällen, wenn der Spieler etwa in einem verschlossenen Behälter eingesperrt ist, ist der momentane Aufenthaltsraum `aRaum` immer in Sichtweite. Räume sind standardmäßig `hell`, so dass überall Licht ist, wenn nichts anderes angegeben wird.

Um einen Raum dunkel zu machen, muss man ihm das Attribut `hell` explizit mit `nicht` abnehmen:

```
Raum Tunnel = {
    Name = "Tunnel";
    Ausgänge = {
        Westen: antworte Tunnelausgang;
        Süden: antworte Tunneleingang;
    }
    Beschreibung = "Dies ist ein pechschwarzer
        Tunnel, der  hier eine Kurve macht und
        von Süden nach Westen läuft.";
    ist nicht hell;
}
```

× `hell` und `Lichtquelle` sind Synonyme für dasselbe Attribut.

. . . . .

Ob es dunkel ist oder nicht, wird in der Variable `dunkel` beschrieben. Diese Variable wird am Ende jedes Zuges automatisch aktualisiert. Wenn also etwas passiert wie

```
mache Fackel nicht Lichtquelle;
entferne Streichholz;
mache Lampenschirm nicht transparent;
```

merkt die Bibliothek das und gibt die neuen Lichtverhältnisse bekannt, entweder mit einer Raumbeschreinung oder mit dem `LibText(Lage, 10)`, der standardmäßig sagt: „Es ist nun stockdunkel hier.“

× Ein Nachteil der automatischen Überprüfung ist, dass es nicht sofort hell oder dunkel wird, sondern erst am Ende des Zuges. Man kann die Lichtverhältnisse aber auch vorher überprüfen indem man die Routine

```
Lichtverhältnisse();
```

in seinem Code aufruft.

× Die genauen Regeln zum Ermitteln der Lichtverhältnisse sind wie in Inform: Es ist hell, wenn das Mutterobjekt des Spielers „Licht bekommt“. Ein Objekt „bekommt Licht“, wenn es das Attribut `Licht` hat, eines einer direkten Tochterobjekte Licht spendet oder es lichtdurchlässig ist und sein Mutterobjekt Licht bekommt. Ein Objekt „spendet Licht“, wenn es selbst das Attribut `Licht` hat, wenn es lichtdurchlässig ist und eines seiner Kinder Licht spendet oder wenn eines seiner Unterobjekte Licht spendet. Lichtdurchlässig sind nicht geschlossene Behälter, Ablagen und transparente Objekte.

Diese Verhältnisse können mit den Routinen `spendet_Licht(obj)` und `bekommt_Licht(obj)` überprüft werden.

× Wer die Dunkelheit in seinem Spiel abschaffen will, kann einfach die Routine `bekommt_Licht` ersetzen, so dass sie immer `wahr` antwortet.

7.11 Die Sinne

Der wichtigste Sinn in einem Adventure ist wohl die Sicht. Beschreibungen von Räumen und Objekten beziehen sich meist auf das, was man sieht, und „untersuche“, eines der meistbenutzten Verben in Textadventures, ist ein Synonym für „schaue an“.

Wenn ein Objekt untersucht wird, dann wird seine `Beschreibung` ausgegeben. Die Beschreibung ist in der Regel ein Text, kann aber auch eine Routine sein. Bei einem `Behälter` oder einer `Ablage` wird zusätzlich der Inhalt gezeigt. Wenn ein Objekt keine Beschreibung und keinen Inhalt hat, so hängt die Ausgabe von den Attributen des Objekts ab. Es wird gesagt, dass das Objekt offen, geschlossen, ein- oder ausgeschaltet ist. Ist bislang immer noch nichts ausgegeben worden so sieht der Spieler „nichts Besonderes“ an diesem Objekt.

Das Untersuchen von Objekten schlägt fehl, wenn es dunkel ist – es sei denn, ein anderes Verhalten wird in `Vorher` implementiert.

Wenn das Objekt der Aufenthaltsraum des Spielers ist, wird die Raumbeschreibung (mit `Lage`) ausgegeben. Die Beschreibungen von Räumen werden in [nicht gefunden] näher erklärt.

. . . . .

Aber auch die anderen Sinne sind mit den Verben `riechen`, `zuhören`, `schmecken` und `fühlen` vertreten. Jedes dieser Verben verlangt ein Objekt, dabei wird vereinfachend angenommen, dass Dinge, die man schmecken und fühlen kann, erreichbar sein müssen, und

Dinge die man riechen und hören kann in Sichtweite sind. (Auch hier werden die Sinne wieder auf die Sicht zurückgeführt. Man kann die Regeln aber ändern, siehe Abschnitt 9.6.)

Das Standardverhalten dieser Befehlen ist allerdings wenig aussagekräftig, der Spieler riecht, hört, schmeckt oder fühlt „nichts Besonderes“. Besondere Sinneswahrnehmungen müssen also in `Vorher` berücksichtigt werden.

Die beiden Befehle `zuhören` und `riechen` können auch ohne Objekte benutzt werden, dann wird der Aufenthaltsraum impliziert.

```
Raum beim_Wasserfall {
    Name = "Beim Wasserfall";
    ...
    Vorher = {
        riechen: stop "Es riecht erdig hier,
            moosig.";

    }
    Vorreaktion = {
        zuhören: stop "Du hörst nur das donnernde
            Tosen das Wasserfalls, sonst nichts.";
    }
}
```

In diesem Fall kann der Spieler sein Transistorradio nicht mehr hören, weil der Wasserfall alles übertönt. Den erdigen Geruch bekommt der Spieler aber nur, wenn er „rieche“ eingibt, ohne Objekt, so dass er einen Harzer Käse immer noch erschnuppern kann, wenn er will.

7.12 Der Spieler

Der Protagonist des Spiels, den der Spieler durch das Abenteuer führt, wird in ZETA durch ein Objekt beschrieben, das Objekt `Spieler`.

Der `Spieler` hat wie alle anderen Objekte eine Position im Objektbaum, meist befindet er sich direkt in einem Raum, er kann aber auch, wie in Abschnitt 7.4 beschrieben, in anderen Objekten sein, die das Attribut `betretbar` haben müssen.

Mit `Mutter(Spieler)` bezieht man sich auf das direkte Mutterobjekt des Spielers, etwa einen Stuhl, auf dem er sitzt oder einfach den Raum, in dem er sich aufhält. Die Variable `aRaum` gibt immer den Raum an, in dem sich der Spieler, eventuell indirekt, aufhält. Sitzt der Spieler also auf einem Stuhl, der auf einem Podest im Konferenzzimmer steht, ist `aRaum` das Konferenzzimmer, obwohl das direkte Mutterobjekt des Spielers der Stuhl ist.

× Es gibt Fälle, in denen `aRaum` nicht sichtbar ist, zum Beispiel, wenn der Spieler in einer geschlossenen Kiste aus nicht durchsichtigem Material ist. Für solche Fälle gibt es die Variable `Sichtgrenze`, die das am nächsten oberhalb des Spielers im Objektbaum befindliche Objekt beschreibt, das nicht durchsichtig, also ein geschlossener Behälter, der nicht `transparent` ist, bezeichnet. Dies ist in der Regel identisch mit `aRaum`, in Sonderfällen aber nicht. Es wird immer die Raumbeschreibung der `Sichtgrenze` ausgegeben.

× Die Dunkelheit ist kein eigener Raum. Wenn sich der Spieler in einem dunklen Raum befindet, ist `aRaum` dieser Raum. Die `Sichtgrenze` ist dann allerdings der Spieler selbst.

. . . . .

Die Variablen `aRaum` und `Sichtgrenze` sollten von Autoren nie explizit geändert werden, dies übernimmt die Bibliothek. Deshalb solte man den Spieler auch nie einfach nur im Objektbaum verschieben. Man kann natürlich Aktionen wie

```
Ausf(gehen, Nord_Objekt);
Ausf(betreten, Tempel);
```

auslösen, um den Spieler in einen benachbarten Raum zu bewegen. Um den Spieler an einen beliebigen Ort zu bringen, ruft man am besten die Routine

```
gehe_zu(obj, b);
```

auf, die den Spieler in das Objekt `obj` bringt und die Variablen `aRaum` und `Sichtgrenze` anpasst und einige andere Dinge überprüft, zum Beispiel ob es am neuen Ort dunkel ist oder nicht. Die Flagge `b` gibt an, ob nach dem Ortswechsel eine Raumbeschreibung ausgegeben werden soll: zwei unterdrückt die Raumbeschreibung, eins schreibt eine ausführliche Beschreibung, so als ob man `Lage` ausführt und null schreibt die „normale“ Raumbeschreibung, wie sie beim Betreten des Raums ausgegeben würde, das heißt je nach Modus und ja nachdem, ob der Spieler den Raum bereits besucht hat, wird eine knappe oder eine ausführliche Bescheribung geschrieben.

In der Regel ist der Zielort `obj` ein Raum, er kann aber auch ein betretbares Objekt sein, das sich in einem Raum oder in anderen betretbaren Objekt befindet. Dass `obj` ein gültiges Objekt ist, muss der Autor sicherstellen. Wenn `obj` ein Sitzmöbel ist, sollte die `Sitzposition` angepasst werden.

× Die Routine kann ein drittes Argument haben, das ein Fahrzeug oder ein Reittier bezeichnet, auf oder in dem der Spieler sitzt, zum Beispiel

```
gehe_zu(Basar, falsch, fliegender_Teppich)
```

In diesem Fall wird nicht der Spieler sondern das Fahrzeug bewegt – und der Spieler natürlich mit ihm. Danach werden die Variablen angepasst wie normal.

▪ ▪ ▪ ▪ ▪

Der Spieler hat natürlich nicht nur ein Mutterobjekt, sondern auch Tochterobjekte, sein Inventar. Genau wie bei Behältern und Ablagen beschreibt seine Eigenschaft `Kapazität` die Anzahl der Objekte, die er tragen kann. Dabei werden am Körper getragene Objekte, die das Attribut `angezogen` besitzen, nicht mitgezählt.

Die Kapazität des Spielers ist standardmäßig 64, der Spieler kann beinahe beliebig viele Gegenstände mit sich herumschleppen. Die Eigenschaft `Kapazität` kann aber beliebig geändert werden, sie muss ein positiver Wert sein oder eine Routine, die einen solchen Wert zurückgibt. (Null und negative Werte sind auch möglich, dann kann der Spieler aber nichts mit sich herumtragen, ihm sind quasi die Hände gebunden.)

Das Inventar des Spielers ändert sich ständig, meist dadurch, dass der Spieler Dinge aufhebt oder wieder ablegt. Der Befehl zum Aufheben von Gegenständen heißt – Überraschung! – `aufheben`. Wenn dieser Gegenstand auf einer Ablage oder in einem Behälter liegt, wird dieser Befehl umgewandelt in `herunternehmen` oder `herausnehmen`, so dass Behälter und Ablagen mit den passenden Pseudobefehlen reagieren können.

In jedem Fall wird das Aufheben in den Befehl `nehmen` umgewandelt. Hier wird geprüft, ob der Spieler ausreichend Tragekapazität hat und ob der Gegenstand überhaupt aufgehoben werden kann. Um das Aufheben eines Gegenstands zu unterbinden, sollte man das `nehmen` abfangen.

Analog dazu wird beim Ablegen eines Gegenstands in einen Raum, einen Behälter oder auf eine Ablage der Befehl `hergeben` aufgerufen. Dieser Befehl macht normalerweise nichts, aber hier kann man das Ablegen eines Gegenstands verhindern, wenn man möchte, und zwar effektiv für alle drei Fälle.

× Das erscheint zwar vielleicht etwas umständlich, dient aber dazu, das Verhalten konsistent zu halten. Nehmen wir einmal an, heiße Kohlen liegen in einem Ofen und können nur mit Asbesthandschuhen angefasst werden. Würde man statt `nehmen`, das übrigens nie direkt vom Parser erzeugt werden kann, den Befehl `aufheben` abfangen, hätte man zwar „nimm Kohle“ unterbunden, „nimm Kohle aus Ofen“ aber nicht. Dasselbe gilt für das Ablegen: Wenn ein Klebestreifen sich nicht ablegen lässt, weil er an der Hand festpappt, unterbindet man mit `hergeben` „lege Klebestreifen ab“ und „lege Klebestreifen auf Tisch“ gleichermaßen.

× Ein Schwachpunkt dieses Modells ist allerdings, das Räume das Ablegen und Herausnehmen von gegenständen nicht mit einem Pseudobefehl behandeln können. Hier muss

man die `Vorreaktion` des Raums bemühen.

. . . . .

`Spieler` ist kein Objekt, sondern eine globale Variable, die auf das jeweilige Spielerobjekt zeigt. Dieses Objekt kann also während des Spiels verändert werden, wenn sich der Spieler zum Beispiel in jemand anderes verwandelt oder einfach vorübergehend den Part einer anderen Person übernimmt.

Die Routine `Spielerwechsel(obj, b)` verwandelt den Spieler in `obj`, dabei wird nicht nur die Variable `Spieler` verändert, sondern auch `aRaum` und `Sichtgrenze` sowie die Schiebekulissen und die Lichtverhältnisse berücksichtigt. Die Flagge `b` gibt an, ob und in welchem Modus eine Beschreibung nach dem Wechsel ausgegeben wird, genau wie bei `gehe_zu`.

× Der momentane `Spieler` ist immer `verborgen`, so dass für ihn keine Sonderregelungen getroffen werden müssen: Er taucht nicht in Raumbeschreibungen auf. Das alte Spielerobjekt bleibt im Spiel und ist nicht mehr `verborgen`. Wenn das nicht so sein soll, muss es explizit aus dem Objektbaum entfernt werden.

× Mit Formen von „ich“ und in der Regel auch „du“ bezeichnet man immer den `Spieler`, so dass ein Objekt, das den Spieler repräsentiert, nicht unbedingt ein eigenes Vokabular haben muss. Ungeachtet der Eigenschaft `Name` des Spielers ist die Ausgabe für das Spielerobjekt immer eine Form von „du“ oder genauer gesagt, das, was die Routine `du_Name`, die natürlich ersetzt werden kann, ausgibt, siehe Abschnitt 8.1.

× Die Bibliothek teilt dem Spieler nie mit, welches Objekt gerade der `Spieler` ist, man kann Verwandlungen also auch unbemerkt vornehmen. Wenn die Metamorphose allerdings offensichtlich ist, sollte man einen Hinweis in die Statuszeile oder die Raumbeschreibung schreiben.

× Das vorgeferitgte Spielerobjekt heißt `Protagonist`. Es hat normalerweise keine besonderen Eigenschaften, außer der Beschreibung, die mit dem Lib-Text Nummer acht für `untersuchen` geändert werden kann. Es besitzt allerdings die wichtigsten Eigenschaften, die jedoch leer definiert sind. Auf diese Weise kann man dem `Protagonisten` nachträglich Eigenschaften wie `Anweisungen` oder `Vorher` geben. Natürlich kann man dieses Objekt auch ersetzen.

7.13 Zeitabhängige Vorgänge

Viele Dinge im Spiel geschehen nicht als unmittelbare Reaktion auf eine konkrete Handlung des Spielers, sie laufen einfach unabhängig im Hintergrund ab. Um solche Ereignisse zu programmieren, gibt es drei ähnliche, in Details aber unterschiedliche Konzepte: Dämonen, Zünder und `Jede_Runde`. Diesen Konzepten ist gemeinsam, dass sie zu einem Objekt gehören und am Ende jeden Zuges ausgeführt werden, nachdem der Spieler seine Handlung ausgeführt hat und unabhängig davon, ob diese ein Erfolg war.

Die Eigenschaft `Dämon` kann eine Routine enthalten, die am Ende des Zuges ausgeführt wird, wenn dieser Dämon aktiv ist. Man kann Dämonen mit

```
Dämon_An(obj);
```

aktivieren und mit

```
Dämon_Aus(obj);
```

deaktivieren. Mit der Routine `Dämon_aktiv(obj)` kann man feststellen, ob der Dämon eines Objekts gerade aktiv ist. Wenn man aktive Dämonen ein- oder inaktive ausschaltet, passiert nichts.

. . . . .

Ein Zünder zählt, einmal aktiviert, eine bestimmte Anzahl von Zügen herunter, bevor er, ebenfalls am Ende eines Zuges, ein Ereignis auslöst, das in der Eigenschaft `Zündung` des Objekts programmiert ist. Um den Zünder zu aktivieren, muss man zusätzlich zum betreffenden Objekt den Zündzeitpunkt angeben:

```
setze_Zünder(obj, Zündzeit);
```

Die `Zündzeit` ist die Anzahl der Züge, in der das Eregnis `obj.Zündung` passiert. Nach jedem Zug wird die verbleibende Zeit um eins verringert, die Zündung wird ausgelöst, wenn dieser Wert negativ wird. Eine Zündzeit von Null löst also ein Ereignis noch im selben Zug aus.

Ein Zünder ist nicht unaufhaltsam, er kann mit

```
lösche_Zünder(obj);
```

wieder gelöscht werden. Die Routine `Restzeit(obj)` gibt an, wieviele Züge bis zur Zündung verbleiben oder –1, wenn die Zündung nicht aktiv ist.

× Die Dämonen und Zünder laufen in einer nicht vorhersagbaren Reihenfolge ab, so dass sich der Autor nicht darauf verlassen kann, dass ein bestimmter Dämon schon gelaufen ist, wenn gerade ein anderen abgearbeitet wird. Auch, wenn man aus einem Dämon heraus einen anderen aktiviert, ist nicht sichergestellt, dass dieser in der selben Runde noch berücksichtigt wird.

× Es können beliebig viele Dämonen und Zündungen definiert werden, ein Objekt kann auch beides besitzen. Es können allerdings immer nur `max_Zünder` Dämonen und Zünder gleichzeitig aktiv sein. Wer mehr benötigt, definiert diese Konstante vor dem Einbinden der *z-Lib* mit einem höheren Wert.

- **Aufgabe 11**

Implementiere eine Schiebetür, die sich mit einer Codekarte öffnen lässt und die einen Zug lang offen bleibt.

- **Aufgabe 12**

Dämonen eigen sich gut, um wiederkehrende Verhaltensmuster bei anderen Personen im Spiel zu implementieren. Wie sähe ein Dieb aus, der sich – wie in *Zork* – durch die Gänge schleicht und seinen Pfad zufällig sucht?

- **Aufgabe 13**

Um die Beschränkung der laufenden Dämonen aufzuheben könnte man nur einen Dämonen laufen lassen, der dann alle anderen Dämonen aufruft, wenn die dazugehörigen Objekte das Attribut `aktiv` haben. Wie?

- **Aufgabe 14**

Die Dämonen und Zünder laufen wie gesagt nicht in einer vorhersehbaren Reihenfolge ab. Das kann unglücklich sein. Erweitere das System aus der vorigen Aufgabe so, dass jedes Objekt eine Eigenschaft `Dämon_Prio` mit Werten zwischen –10 und 10 haben kann, so dass Dämonen mit einer höheren Priorität zuerst laufen.

. . . . .

Zusätzlich zu diesen beiden Eigenschaften, die erst explizit aktiviert werden müssen, wird am Ende des Zuges für jedes Objekt, das sich in Sichtweite des Spielers befindet, automatisch die Eigenschaft `jede_Runde` aufgerufen, wenn es sie besitzt.

Da das Objekt immer in der Nähe des Spielers ist, wenn `jede_Runde` aufgerufen wird, kann man diese Routine gut dazu verwenden, um einfache Personen zu implementieren, die nur agieren, wenn der Spieler gerade mit ihnen im Raum ist oder

× Man kann die Sichtbarkeit von Objekten künstlich erweitern, mehr dazu in Abschnitt 9.6. Wenn die Bibliothek die Sichtbarkeit prüft, um zu ermitteln, für welche Objekte

`jede_Runde` aufgerufen werden soll, hat die Variable `Kontext` den Wert `k__jede_Runde`.

Die globale Variable `Züge` enthält die Anzahl der vom Spieler durchgeführten Spielzüge. Nicht verstandene Eingaben und Meta-Befehle wie Laden und Speichern werden bei dieser Zählung nicht berücksichtigt. Es gibt in der Bibliothek kein System, um die Uhrzeit zu bestimmen. Ein solches System ist allerdings schnell implementiert, wer will kann die Erweiterung `"uhrzeit.zin"` einbinden.

Schließlich wird, sozusagen als Überdämon, der stets aktiv ist und nicht an die Sichtbarkeit von Objekten gebunden ist, die Eigenschaft `jede_Runde` aller Lib-Objekte ausgeführt. Hier kann man nun all das abfangen, was mit den anderen drei Konzepten (oder Kombinationen aus ihnen) nicht möglich war. Dies ist auch der letzte Ausweg, wenn man die Reihenfolge von Dämonen beachten muss.

Der genaue Ablauf nach Ende jeden Zuges ist wie folgt:

- Alle Dämonen und Zeitzünder werden aufgerufen.
- `Jede_Runde` wird für sichtbare Objekte ausgeführt.
- `Jede_Runde` aller Lib-Objekte wird ausgeführt.
- Die Objekte beim Spieler bekommen das Attribut `bewegt`.
- Die globale Variable `Züge` wird aktualisiert.

7.14 Der Anfang und das Ende des Spiels

Wie bereits in Abschnitt 6.1 beschrieben, wird zu Beginn des Spiels einmalig die Routine `Anfang` aufgerufen. Hier wird in der Regel ein Begrüßungstext ausgegeben, in dem der Spieler erfährt, wer er ist und was er machen soll. Jedes Spiel muss diese Routine `Anfang` definieren, auch, wenn kein Text ausgegeben werden soll.

Vor dieser Routine können Objekte initialisiert werden, dazu gibt es die Eigenschaft `init`, die jedes Objekt besitzen kann. Diese Eigenschaft ist eine Routine und sollte in der Regel keine Text ausgeben. Hier kann man Voraussetzungen für das neue Spiel festlegen, zum Beispiel könnte man einen Stoß Karten mischen oder die Kombination für einen Safe festlegen:

```
init = {
    selbst.Kombination = Zufall(999);
}
```

Dies könnte man natürlich auch in der Routine `Anfang` machen, aber direkt beim Objekt ist es übersichtlicher. Vor den `init`-Routinen der einzelnen Objekte wird `init` für alle Lib-Objekte ausgeführt.

Nachdem der Anfangstext ausgegeben wurde, wird das „Spielbanner“ gedruckt. Es enthält den Titel des Spiels, wie er in der Konstante `Story` definiert wurde und den Untertitel, das `Copyright`. Es folgt eine Zeile, die Seriennummer, Release und die verwendete Version der z-Lib anzeigt. Vor und nach diesen technischen Daten können optional die Lib-Texte `Untertitel` eins und zwei ausgegeben werden.

Folgende Definition:

```
Konstante Story = "Den Tod im Nacken";
Konstante Copyright = "Eine interaktive Hetzjagd,
    (C) 2004 Isabelle Levasseur";

LibObjekt in Zetalib {
    Texte = {
        lokal n;
        Untertitel: wenn (n==2) stop
            "\p(Wer nicht weiter weiß, sagt HILFE.
            Wer keine Lust mehr hat, sagt ENDE.)";
    }
}
```

gibt also zu Beginn des Spiels folgendes Spielbanner aus:

> *Den Tod im Nacken*
> Eine interaktive Hetzjagd, (C) 2004 Isabelle Levasseur
> Release 1, Seriennr. 040628, z-Lib 1.0
>
> (Wer nicht weiter weiß, sagt HILFE. Wer keine Lust mehr hat, sagt ENDE.)

× Mit der Routine `Spielbanner(lang)` kann man das Spielbanner auch später ausgeben lassen. Nur wenn `lang` gesetzt ist, werden die Zusatztexte zu `Untertitel` wie zu Beginn des Spiels ausgegeben. Der Befehl `meta__Version` zeigt nur das kurze Spielbanner, also `Story`, `Copyright` und technische Daten an.

. . . . .

Nach dem Aufruf von `Anfang` muss die Bibliothek wissen, wo der Spieler sein Abenteuer beginnen soll. Dazu muss entweder die Variable `aRaum` auf den Anfangsraum gesetzt werden oder das Objekt `Spieler` muss in das passende Mutterobjekt verschonen werden. Welche Methode man benutzt, ist letztendlich egal, nur sollte man nur eine dieser Methoden verwenden.

× Der Anfangsort des Spielers muss nicht unbedingt ein Raum sein, manche Spiele lassen den Spieler in seinem Bett aufwachen, in diesem Fall würde man den Spieler ins Bett verschieben und die `Sitzposition` auf `liegen` setzen. Es ist auch möglich, den Spieler im Dunkeln, das heißt in einem dunklen Raum anfangen zu lassen.

×

. . . . .

Irgendwann kommt das Ende des Spiels. Der Spieler erreicht sein Ziel und bringt die Geschichte zu einem guten Ende. Oder – was weitaus häufiger passiert – er scheitert, was in so genannten klassischen Spielen soviel heißt wie: er stirbt.

Wann stirbt oder gewinnt der Spieler? Genau dann, wenn die globale Variable `Ende` einen wahren, also von null verschiedenen Wert hat. Die Konvention in ZETA ist, dass ein erfolgreiches Ende mit einen postiven Wert von `Ende`, ein schlimmes Ende mit einem negativen bedacht wird. Der Autor kann hier, wenn er will, zwischen mehreren möglichen Siegen und Niederlagen unterscheiden.

Anders als im richtigen Leben ist, was nach dem Tod (beziehungsweise nach dem Ende) passiert, genau geklärt. Zunächst wird der `LibText(Spielende, 0)` ausgegeben. Per Default ist das nichts, manche Spiele geben hier aber einen Abriss der erreichten Punkt aus. Man kann hier auch die Variable `Ende` zurück auf null setzen um den Spieler im letzten Moment dem Tod von der Schippe springen zu lassen.

Danach gibt das Spiel den `LibText(Spielende, Ende)` aus, und zwar einer alten Tradition folgend umrahmt von Sternchen und in Fettdruck. Standardmäßig wird bei einem positiven Wert von `Ende`

*** Du hast gewonnen. ***

ausgegeben, bei einem negativen

*** Du hast verloren. ***

Genauere Meldungen zu den jeweiligen Werten von `Ende` müssen durch Abfangen der Lib-Texte implementiert werden. Wer die Textauszeichnungen nicht haben möchte, kann die Routine `Ende_verkünden` durch folgende ersetzen:

```
Routine Ende_verkünden {
    "\p", LibText(Spielende, Ende), "\p";
}
```

Schießlich erhält der Spieler die Möglichkeit, das Spiel von vorn zu beginnen, einen Spielstand zu laden, den letzten Zug rückgängig zu machen oder das Spiel zu beenden.

× Die Eingabe des Spielers wird bei dieser Frage – genau wie bei `Ja_oder_Nein` – nur grob geparst, und zwar so, dass `aBefehl` den ersten Wert annimmt, den es aufgrund des ersten Worts haben könnte, alle weiteren Wörter bleiben unberücksichtigt. Das ist für die Meta-Befehle, die ja meist nur aus einem Wort bestehen, gut genug.

× Wenn `aBefehl` keiner der vier oben genannten Meta-Befehle ist, wird der `LibText(verschiedenes, 4)` ausgegeben. Wer will, kann hier zusätzliche Optionen nach Spielende einbauen.

- **Aufgabe 15**

Manche Spiele bieten dem Spieler ein Nachwort an, meist um auf lustige Dinge hinzuweisen, die der Spieler eingeben kann. Implementiere das.

7.15 Die Statuszeile

Üblicherweise befindet sich oben am Bildschirm ein Bereich, der invers gedruckt ist und der allgemeine Informationen über das Spiel enthält, die so genannte Statuszeile. Wenn nichts anderes angegeben wird, ist die Statuszeile eine Zeile hoch und in der linken oberen Ecke steht einfach der Name des Aufenthaltsraums, eventuell gefolgt von einem anderen Objekt, in oder auf dem sich der Spieler gerade befindet.

Das Aussehen der Statuszeile kann aber verändert werden, dazu gibt es zwei Möglichkeiten.

Die einfachere ist, die Höhe der Statuszeile mit der Variable `Statuszeilen` festzulegen und den Inhalt der Statuszeile in den Lib-Texten zum Pseudobefehl `Status` mit den Nummern eins bis drei. Der erste Text wird linksbündig geschrieben, der zweite zentriert und der dritte rechtsbündig. Zeilenumbrüche in den Texten sind möglch, wenn es aber mehr Zeilen als `Statuszeilen` gibt, wird der Text entsprechend gekürzt.

Die komplizierte Methode ist, die Routine `Statuszeile` zu ersetzen und eine eigene zu schreiben, in der dann von Hand alle Texte in der Statuszeile positioniert werden. Dabei muss beachet werden, dass das obere Fenster im Interpreter nicht automatisch invers dargestellt ist. Wer eine eigene Statuszeile programmieren will, findet das nötige Rüstzeug dazu in Abschnitt 3.4.

- **Aufgabe 16**

Schreibe eine Liste der möglichen Ausgänge rechts oben in die Statuszeile.

8 Beschreibungen

8.1 Namen von Objekten und Räumen

Jedes Objekt, das im Spiel auftaucht, muss einen Namen besitzen, der dann in den vom Parser generierten Meldungen, zum Beispiel „Du siehst nichts Besonderes an dem silbernen Esslöffel“, eingesetzt wird.

Um den Namen des Objekts korrekt ausgeben zu können, muss man die Eigenschaften `Name` und `Genus` definieren. Der `Name` ist ein Text oder eine Routine, um Text auszugeben. Der `Genus` kennzeichnet das grammatische Geschlecht, das zum Namen passt:

`m`	*Maskulinum*, männlich
`f`	*Femininum*, weiblich
`n`	*Neutrum*, sächlich
`p`	*Plural*, Mehrzahl

Mit diesen beiden Informationen kann die Bibliothek das Objekt nun immer mit dem richtigen Artikel ausgeben.

× Da Artikel und Endungen für alle in der Mehrzahl stehenden Substantive („die Nägel“) gleich sind, muss man nicht zwischen Plural männlich, Plural weiblich und Plural sächlich unterscheiden.

Im Deutschen werden Adjektive und auch Substantive je nach Fall gebeugt. Wenn also ein Name Adjektive enthält, so muss die Bibliothek wissen, wo solche Beugungen eingefügt werden. Dies wird mit so genannten *Kurzstrings* (siehe Abschnitt 1.9) gemacht, die je nach Genus des Objekts und Fall die passenden Werte haben.

Für Adjektivendungen benutzt man den Kurzstring `{e}`, der einfach ans ungebeugte Wort angehängt wird:

```
Name = "golden{e} Taschenuhr";
Name = "klein{e} bunt{e} Schnipsel";
Name = "eigenartig wässrig{e} Weintrauben";
```

Adverbien, die Adektive näher beschreiben, müssen nicht gebeugt werden und bekommen daher auch keine Endung.

Manche Substantive, die im Plural stehen, bekommen im Dativ ein „-n“ angehängt. Dies wird

mit `{n}` berücksichtigt:

```
Name = "Lagerregale{n}";
Name = "mürisch dreinschauend{e} Zwerge{n}";
```

Für Wörter lateinischer Herkunft, die in allen Fällen außer dem Nominativ die Endung „-en“ haben, setzt man `{en}`:

```
Name = "langhaarig{e} Student{en}";
```

Oft wird an Substantive im Genitiv ein „-s“ oder „-es“ angehängt. Hierfür setzt man `{s}` und `{es}`:

```
Name = "Apfel{s}";
Name = "Ziegelhaus{es}";
```

Der Genitiv wird allerdings nur in seltenen Fällen und von der Bibiothek nie zur Ausgabe verwendet.

Dieses Schema funktioniert gut für die meisten Fälle. Einige Objekte benötigen aber spezielle Regeln, die durch Sonderkonstanten, die zum `Genus` addiert werden, beschrieben werden. Mit `Eigenname` bewirkt man, dass ein Objekt wie ein Eigenname behandelt wird, also immer ohne Artikel ausgegeben wird:

```
Name = "Tamara Durango";
Genus = f + Eigenname;

Name = "Franks alt{e} Kaffeetasse";
Genus = f + Eigenname;
```

Wenn ein Objekt immer mit dem bestimmten Artikel anstelle des unbestimmten ausgegeben werden soll, setzt man `immer_bestimmt`:

```
Name = "Smaragd der Ewigen Weisheit";
Genus = m + immer_bestimmt;
```

Mit `einige` oder `etwas` wird anstelle des unbestimmten Artikels „einige“ für Mehrzahlobjekte, „etwas“ für nicht quantifizierbare Objekte in der Einzahl benutzt:

```
Name = "Kieselsteine";
Genus = p + einige;

Name = "Weihwasser";
Genus = n + etwas;
```

× Die Eigenschaften `Name` und `Genus` können natürlich eine Routine sein. Die Kurzstrings und Sonderkonstanten gelten dann genau wie bei einem Text.

× In speziellen Fällen muss man den Namen von Hand anpassen, damit er in allen Fällen korrekt ausgegeben wird. Dann muss man auf einige globale Variablen zurückgreifen, die den momentanen Ausgabemodus enthalten.

Die Variable `__Kasus` beschreibt den Fall, in dem der Name ausgegeben werden soll und enthält einen der konstanten Werte `Nominativ`, `Genitiv`, `Dativ` oder `Akkusativ`. Die Variable `__Genus` enthält den `Genus` des Objekts ohne Sonderkonstanten, das heißt reduziert auf `p`, `m`, `f` oder `n`. `__Artikel` beschreibt, ob das Objekt mit bestimmtem (`def_Art`), unbestimmtem (`indef_Art`) oder ohne Artikel (`ohne_Art`) ausgegeben werden soll.

Zur Ausgabe stehen weiterhin einige Felder zur Verfügung. Sie enthalten Vokabeln (müssen also mit `Vokabel(x)` ausgegeben werden) und sind mit

```
__Kasus*4 + __Genus
```

indiziert. Diese Felder enthalten Artikel und Endungen für Adjektive und Pronomina, ihre Bezeichner sind lang, aber weitestgehend selbsterklärend:

```
def_Artikel
indef_Artikel
indef_Art_Endung
ohne_Adj_Endung
def_Adj_Endung
indef_Adj_Endung
personal_Pron
possessiv_Pron_Endung
demonstrativ_Pron_Endung
```

All diese Felder enthalten Vokabeln, keine Texte. Zum Ausgeben muss daher die Routine `Vokabel` benutzt werden.

× Der Name des Objekts `Spieler` ist unabhängig davon, welches Objekt dies gerade ist, die passende Form von „du“. Es sei denn, die Routine `du_Name()` wird ersetzt.

. . . . .

Diese Regeln zur Ausgabe werden natürlich nicht nur von der Bibliothek benutzt, man kann sie auch selbst benutzen. Die Routinen

```
das_Objekt(o, k);
ein_Objekt(o, k);
```

```
kein_Objekt(o, k);
Objekt_ohne_Art(o, k);
```

geben das Objekt `o` im Fall `k` mit den passenden Artikeln aus. Da die Angabe des Kasus im Fließtext etwas umständlich ist, gibt es für die ersten beiden Routinen Abkürzungen, die den Kasus bereits beinhalten:

```
der(o);    des(o);    dem(o);    den(o);
ein(o);    eines(o);  einem(o);  einen(o);
```

× Dies sind die männlichen Formen der Artikel – nur für diesen Genus sind alle Fälle eindeutig unterscheidbar. Man denkt hier also besser nicht an *das* Objekt, sondern *den* Gegenstand.

Weitere Funktionen erlauben die korrekte Ausgabe von Verben, je nachdem, ob ein Objekt im Plural steht oder nicht.

`v_ist(o)`	„ist“ oder „sind“
`v_hat(o)`	„hat“ oder „haben“
`v_wird`	„wird“ oder „werden“
`v_t(o)`	„t“ oder „en“
`v_tt(o)`	„t“ oder „n“
`v_allg(s, p, o)`	`s` oder `p`

Die vorletzten beiden Routinen sind für Verbendungen gedacht, die letzte erlaubt eine absolut freie Wahl der Strings `s` und `p` für Singular und Plural:

```
der(o), " steh", v_t(), " im Weg.";
der(o), " müffel", v_tt(), " etwas.";
der(o), " ", v_allg("fällt", "fallen"),
    " aus dem Rahmen.";
```

× Diese Beispiele zeigen eine weitere Regel: Wenn bei Routinen zur Objektausgabe das Objekt fehlt, wird das zuletzt verwendete, in diesen Fällen also `o`, eingesetzt. Die runden Klammern müssen natürlich trotzdem stehen, auch wenn sie leer sind.

× Die Sätze oben sind so noch nicht ganz richtig, sie fangen nämlich klein an. Die Routine

```
Groß(Routine, Argumente);
```

ruft die Routine auf und wandelt den ersten Buchstaben der Ausgabe in einen Großbuchstaben um:

```
Groß(der, Mutter(o)), " ", v_ist(), " noch zu!";
```

!! Hier muss man aufpassen, dass die Routine und die Argumente einzeln an `Groß` übergeben werden, und nicht etwa der Rückgabewert der Routine. Etwas wie:

```
Groß(dem(o)), " war alles egal.";
```

erzeugt einen Laufzeitfehler.

× Einige Präpositionen und bestimmte Artikel verschmelzen zu einem Wort, zum Beispiel „im“, „aufs“ oder „zur“. Wenn man diese Verschmelzungen berücksichtigen möchte, kann man die Routine

```
Präp_das_Objekt(p, o, k);
```

benutzen: Es wird die Präposition `p`, danach das Objekt `o` im Kasus `k` mit bestimmtem Artikel ausgegeben, dabei werden eventuelle Verschmelzungen berücksichtigt. Auch hier gibt es wieder Abkürzungen, so dass

```
Präp_den('in', aObj);
Präp_dem('in', aObj);
```

nun ein sprachlich flüssiges „ins Meer“ oder „im Meer“ ausgeben. In den meisten Fällen bleiben Präposition und Artikel aber getrennt: „in den Ozean“.

!! Wichtig ist hier, dass die Präpositionen als Vokabeln zwischen Hochkommas, nicht als Texte angegeben werden. Zum Ermitteln der Verschmelzungen werden die Synonymdefinitionen des Parsers verwendet.

. . . . .

Ein Sonderfall sind die Namen von Räumen. In Raumbeschreibungen oder in der Statuszeile werden sie wörtlich ausgegeben, etwa „In einem Kornfeld“ oder „Billardzimmer“.

Dort wird man diese Namen auch am häufigsten lesen. Da Räume, wie alle Objekte, Vokabular haben können, muss es auch Ausgaberegeln für die Raumnamen in verschiedenen Fällen geben. Wenn ein Raum die Eigenschaft `Genus` hat, dann wird der Raumname genau wie der Name eines Objekts angegeben. Das heißt, Adjektivendungen müssen mit `{e}` markiert werden. Wenn kein `Genus` definiert ist, wird einfach „dieser Ort“ benutzt, so dass die Antwort auf „nimm Kornfeld“ wäre:

> Du kanst diesen Ort nicht nehmen.

Nicht besonders elegant vielleicht, aber zumindest sprachlich korrekt. Wenn der Raumname als Titel für eine Raumbeschreibung oder die Statuszeile verwendet werden soll, wird der

`Kontext` auf `k__Raumtitel` gesetzt. Damit können dann flexible Ausgaberoutinen geschrieben werden:

```
Name = {
    wenn (Kontext == k__Raumtitel)
        stop "In einem immensen Kornfeld";
    "immens{e} Kornfeld";
}
Genus = n;
```

8.2 Listen von Objekten

In Beschreibungen von Räumen und des Inventars werden mehrere Objekte hintereinander als Satz aufgelistet. Zum Schreiben der Liste gibt es ein Hilfsobjekt `Liste`, dessen Name den momentanen Inhalt der Liste ausgibt, wie zum Beispiel in der Eigenschaft `Inhalt` eines Behälters:

```
"In der Schubkarre siehst du ", einen(Liste), ".";
```

Oft ist der Inhalt der Liste bereits vorgegeben. Der Autor kann aber auch selbst das Listenobjekt benutzen um eigene Listen zu erstellen. Die Routine

```
erstelle_Liste(obj, attr, add)
```

erstellt eine Liste aller Tochterobjekte von `obj`, die das Attribut `attr` haben. Ist `add` gleich null, wird eine neue Liste angelegt, ansonsten wird die neue Liste einfach an die alte angehängt. `attr` kann auch negativ sein, dann werden alle Objekte berücksichtigt, die dieses Attribut nicht haben. Ist `attr` null, so wird alles in die Liste mit aufgenommen, ungeachtet des Attributs. Die Routine antwortet mit der Anzahl der neu zur Liste hinzugefügten Objekte. Die Anzahl der Objekte in der Liste steht in `Liste.Anzahl`.

× Mit `erstelle_globale_Liste(attr, add)` kann man alle Objekte, nicht nur die Töchter eines einzelnen Objekts berücksichtigen.

× Man kann auch einene Routinen zum Anlegen der Liste schreiben. Mit `setze_auf_Liste(obj)` wird ein Objekt auf die Liste gesetzt. Mit `lösche_Liste()` kann man die Liste löschen.

Das Erstellen der Liste erzeugt noch keine Ausgabe, es speichert nur die zu listenden Objekte zwischen. Um die Liste auszugeben, muss man den Namen des Objekts `Liste` ausgeben.

. . . . .

Das Aussehen der Liste lässt sich in einem bestimmten Rahmen durch die Eigenschaft `Liste.Stil` beeinflussen. Dieser Listenstil ist eine binäre Kombination aus mehreren Flaggen. So wird zum Beispiel „oder“ statt „und“ ausgegeben, wenn `l__oder` gesetzt ist. Setzen kann man diese Flagge mit:

```
Liste.Stil |= l__oder;
```

. . . . .

Gerade in Inventarlisten, aber gelegentlich auch in Raumbeschreibungen, werden gerne Zusatzinformationen zu den Objekten ausgegeben, die in der Regel in Klammern angehängt werden:

> Du hast einen Rucksack und eine Taschenlampe (eingeschaltet) und etwas Kleingeld (1 Euro und 85 Cent) bei dir. In deinem Rucksack sind eine Flasche Himbeergeist (halbvoll), drei Landjäger und ein Anorak.

Das ist natürlich ein extremes Beispiel, und standardmäßig schreibt die Bibliothek in Listen nichts als die Objektnamen. Wenn im Listenstil aber `l__Info` gesetzt wird, dann wird zu jedem Objekt, das die Eigenschaft `Zusatzinfo` besitzt, diese Eigenschaft aufgerufen. Das kann ein Text sein, in der Regel ist es aber eine Routine:

```
Zusatzinfo = {
    jenach (selbst.Füllstand) {
        2: " (voll)";
        1: " (halbvoll)";
        sonst: " (leer)";
    }
}
```

Zusätzlich dazu wird die `Zusatzinfo` der Lib-Objekte aufgerufen, um allgemein gültige Regeln für diese zusätzlichen Informationen schreiben zu können. Dann wird das momentan betrachtete Objekt als Argument übergeben:

```
Zusatzinfo = {
    lokal o;
    wenn (o ist Lichtquelle) stop " (Licht spendend)";
    wenn (o ist schaltbar) {
        wenn (o ist eingeschaltet) stop " (eingeschaltet)";
        stop " (ausgeschaltet)";
    }
    wenn (o ist schließbar) {
        wenn (o ist abgeschlossen) stop " (abgeschlossen)";
        wenn (o ist geschlossen) stop " (geschlossen)";
        stop " (offen)";
    }
```

```
}
```

× Das geschieht nur für Objekte, deren `Zusatzinfo` keinen wahren Wert zurückgegeben hat, so dass diese Eigenschaft bei Objekten auch benutzt werden kann, wenn die allgemeinen Regeln nicht zutreffen sollen.

× Die Flagge `l__Info` ist sowohl beim Inventar als auch bei Raumbeschreibungen und Inhaltslisten von Räumen, Ablagen und Behältern gesetzt.

× Während des Schreibens von Inventarlisten ist der `Kontext k__Inventar`, beim Schreiben von Inhaltslisten `k__Inhalt`.

. . . . .

Innerhalb einer Liste kann man Objekte gruppieren. Um ähnliche Objekte immer nebeneinander stehen zu lassen, kann man ihnen die Eigenschaft `Listengruppe` geben. Alle Objekte, die dieselbe Listengruppe haben, werden in einer Liste nebeneinander geschrieben.

Die Gruppierung der Objekte kann auch ein eigenes Format haben. Wenn die Flagge `l__gruppieren` im Listenstil gesetzt ist und die gemeinsame Listengruppe ein Objekt ist, wird anstelle der einzelnen Objekte der Name dieses Gruppenobjekts ausgegeben. So kann man aus

> ein Matchbox-Auto, einen Apfel, eine Birne und ein paar Weintrauben

folgendes machen:

> ein Matchbox-Auto und etwas Obst

Nun weiß der Spieler allerdings nicht, welches Obst gemeint ist. Die Liste der Objekte in dieser Gruppe wird durch das Objekt `Unterliste` repräsentiert, das auch, genau wie die `Liste`, mit den gängigen Ausgaberegeln ausgegeben werden kann:

```
Objekt Obstgruppe = {
    Name = {
        "Obst – es handelt sich um ",
            einen(Unterliste), " – ";
    }
    Genus = n + etwas;
}
```

Im Normalfall möchte man die Unterliste im selben Fall und mit dem selben Artikel (unbestimmt oder bestimmt) ausgeben wie die Hauptliste. Dies geschieht mit der Routine

`Gruppenliste()`:

```
Objekt Obstgruppe = {
    Name = {
        "Obst (", Gruppenliste(), ")";
    }
    Genus = n + etwas;
}
```

Für Objekte, die die Eigenschaft `Anzahl` besitzen, kann man im `Genus` die Konstante `Zahlenangabe` verwenden. Wenn `Anzahl` größer als eins ist, wird dann das passende Zahlwort eingesetzt. Zum Beispiel erzeugt

```
Name = {
    "Briefmarken (", Gruppenliste(), ")";
}
Genus = p + Zahlenangabe;
Anzahl;
```

folgende Ausgabe:

> drei Briefmarken (die Blaue Mauritius, die schwarze Ein-Kreuzer-Marke aus Bayern und eine Wohlfahrtsmarke)

Damit dies funktioniert, müssen Objekte, die Listengruppen repräsentieren, die Eigenschaft `Anzahl` haben. Der Wert von `Anzahl` wird automatisch beim Erzeugen der Liste aktualisiert.

× Das Gruppenobjekt ist nur ein Hilfsobjekt, das kein Vokabular hat und nirgendwo im Spiel auftauscht.

× Weil niedrige Zahlen von ZETA oft als Objekte interpretiert werden, sollten Listengruppen, die Objekte ohne eigenes Format einfach nur gruppieren, am besten negative Werte haben. Die Gruppierung erfolgt immer, auch wenn `l__gruppieren` nicht gesetzt ist.

× Natürlich kann `Listengruppe` auch eine Routine sein, die mit einem Objekt oder einem anderen Wert antwortet. Nur wenn null geantwortet wird, werden die Objekte nicht geordnet.

× Die Einträge in der Liste werden im Feld `Listenfeld` abgelegt, das `Liste.Anzahl` Einträge hat, die bei null beginnen. Die `Unterliste.Anzahl` Einträge der momentan betrachteten Untergruppe stehen im Feld ab dem Eintrag mit dem Index `Unterliste.Startwert`.

× Während eine Untergruppe geschrieben wird, ist die Eigenschaft `Inhalt` der Liste gleich dem ersten Objekt der Unterliste, ansonsten ist `Liste.Inhalt` immer null. So kann der `Name` gegebenenfalls für Gruppen angepasst werden. `Unterliste.Inhalt` enthält dann das Objekt mit dem Gruppennamen für die Unterliste.

× Ein Problem, wenn man im Namen des Gruppenobjekts Nebensätze verwendet, ist, dass man nicht weiß, ob zum Schluss ein Komma steht oder nicht. Man kann zwar folgendes prüfen

```
(Liste.Anzahl ==
    Unterliste.Startwert + Unterliste.Anzahl)
```

um zu sehen, ob die Gruppe das letzte Element der Hauptliste ist, aber selbst dann weiß man nicht, ob auf den Namen der Liste im Satz ein Satzzeichen, etwa ein weiteres Komma oder gar ein Punkt, folgt oder ob der Satz einfach weitergeht, also ein Abschlusskomma in der Untergruppe gesetzt werden muss. Deshalb ist es sicherer Klammern zu verwenden, auch wenn Kommas oder Gedankenstriche schöner wären.

8.3 Mehrere gleiche Objekte

Gleiche Objekte, die der Spieler einzeln handhaben kann, können in Listen zu einem „Plural-Objekt“ zusammengefasst werden. Alle Objekte, deren Eigenschaft `Plural` auf dasselbe Objekt verweisen, werden in Listen durch dieses Objekt beschrieben.

Idealerweise benutzt man für die gleichen Objekte eine Klasse und stellt dem Plural eine Zahlenangabe voran:

```
Klasse Murmel {
    Name = "bunt{e} Murmel";
    Genus = f;
    Plural = Murmeln;
    ...
}

Objekt Murmeln {
    Name = "Murmeln";
    Genus = p + Zahlenangabe;
}
```

× Natürlich müssen die Objekte auch in der Merhzahl ansprechbar sein, damit der Spieler etwas wie „lege drei Murmeln hin“ eingeben kann. Das wird in [nicht gefunden] beschrieben.

× Wenn Untergruppen von Pluralobjekten angelegt werden sollen, muss die Eigenschaft `Listengruppe` bei den einzelnen Objekten und beim Pluralobjekt gesetzt werden.

× Bei Untergruppen von Pluralobjekten wird die Anzahl aufsummiert – es wird also etwas wie „acht Bälle (fünf rote, ein gelber und zwei blaue)“ ausgegeben, wenn man beim Pluralobjekt `Zahlenangabe` definiert.

- **Aufgabe 17**

Implemenmtiere die verschiedenfarbigen Bälle.

- **Aufgabe 18**

Implementiere eine Klasse für Lampions, die man an- und ausmachen kann, so dass leuchtende und dunkle Lampions getrennt gelistet werden.

- **Aufgabe 19**

Erweitere die Klasse so, dass die Lampions verschiedene Farben haben, wenn sie leuchten und nach Farben getrennt aufgelistet werden.

9 Der Parser

Als Parser bezeichnet man den Teil des Programms, der den vom Spieler eingegebenen Text analysiert. Die Bibliothek hat bereits umfangreiche Definitionen und Routinen, um die Eingabe des Spielers zu interpretieren. Dabei gibt es neben der rein grammatischen Ebene auch immer Verknüpfungen mit der Semantik, wenn zum Beispiel zwischen mehreren Objekten ein sinnvolles ausgewählt werden soll oder wenn andere Personen im Spiel angesprochen werden sollen, deren Satzbau nicht den allgemein gültigen Regeln gehorcht.

9.1 Verben

Die Konvention ist, dass der Spieler Imperativsätze eingibt, üblicherweise in der zweiten Person: „Nimm das Schwert und untersuche es“ zum Beispiel. Das bedeutet, dass jeder gültige Satz mit der Imperativform eines Verbs anfängt. Vom Standpunkt des Parsers betrachtet ist das angenehm: Das erste Wort bestimmt, welche Strukturen ein nachfolgender Satz haben kann. Und so ist auch die Arbeitsweise des Parsers von ZETA: Wenn ein Verb gefunden wurde, geht er mögliche Satzmuster durch, bis eins auf den eingegebenen Satz passt.

Letzten Endes wird jeder gültige Satz in einen Befehl umgewandelt, mit dem das Weltmodell dann arbeitet. Die Eingabe

> Zwerg, gib dem Troll die kleine Axt

wird nach erfolgreicher Analyse umgewandelt in folgende Variablen:

```
Akteur == Zwerg
aBefehl == geben
aObj == Troll
aObj2 == kleine_Axt
```

Und so ist ...

× Es wäre schön, wenn der Parser auch Sätze wie „die Tür aufschließen“ verstünde, bei denen das Verb als Infinitiv hinten anhängt. Im Moment kann die ZETA-Bibliothek dies allerdings nicht.

9.2 Verben

9.3 Satzmuster

9.4 Objekte

9.5 Parser-Routinen

9.6 Gültigkeit von Objekten

Teil IV:

ZETA für Fortgeschrittene

10 Weitere Funktionalitäten

10.1 Routinen mit beliebig vielen Argumenten

In der Z-Maschine ist die Anzahl der Argumente, die beim Aufruf einer Routine übergeben werden, nicht festgelegt. Wird eine Routine mit weniger Argumenten aufgerufen, als angegeben, so bekommen die restlichen Argumente den Wert 0. Werden mehr Argumente angegeben als lokale Variablen vorhanden sind, so verfallen sie.

Funktionsaufrufe wie `f()` und `f(0)` sind also gleich. Vielleicht möchte man sie aber unterscheiden. Dazu gibt es die Systemfunktion `Argumente()`, die die Anzahl der übergebenen Argumente in der jeweiligen Routine angibt. Dieser Wert wäre null für `f()` und eins für `f(0)`.

ZETA erlaubt es, bei der Deklaration der Routine eine variable Anzahl von Argumenten anzugeben. Dazu setzt man als letztes Argument anstelle eines Bezeichners drei Punkte `...` als Auslassungszeichen:

```
Routine istGleich(x, ...) {
    ...  // wirkliches Auslassungszeichen;
}
```

Die an die Routine übergebenen Argumente haben keine Bezeichner, deshalb muss man sie mit den beiden Systemfunktionen

```
Wert = leseArg(n);
schreibeArg(n, Wert);
```

ansprechen. `n` ist hier der Index aller Argumente und es wird bei eins angefangen zu zählen. In der Routine `istGleich` wäre `leseArg(1)` gleich dem Wert von `x`. Maximal können sieben Argumente an eine Funktion übergeben werden.

Das Zugreifen auf Argumente über deren Index erlaubt einiges flexibles Programmieren, zum Beispiel das Prüfen auf Gleichheit für mehrere Werte:

```
Routine ist_gleich(x, ...) {
    lokal n = Argumente();
    solange (n > 1) {
        wenn (leseArg(n--)==x) antworte wahr;
    }
    antworte falsch;
}
```

Oder das Folgende:

```
Routine max(...) {
    lokal n = Argumente(), m;
    wenn (n) {
        m = leseArg(n--);
        solange (n--)
            wenn (leseArg(n) > m) m = leseArg(n);
    }
    antworte m;
}
```

Hier wird das Maximum über alle angegebenen Werte gebildet. Es können natürlich in der Routine weitere lokale Variablen mit `lokal` definiert werden.

× Mit den Funktionen `leseArg` und `schreibeArg` kann man nicht nur auf die Argumente einer Routine zugreifen. Mit Indizes von acht bis fünfzehn greift man auf weitere lokale Variablen zu, wenn sie definiert sind, mit Indizes von sechzehn bis 255 auf globale Variablen. Das sollte aber besser vermieden werden, denn diese Variablen haben ja alle feste Bezeichner. Mit dem Index null zieht man einen Wert von Stapelspeicher, und das sollte auf keinen Fall gemacht werden, da sonst die Anweisungen durcheinanderkommen und die *Z*-Maschine wahrscheinlich in einen irreparaben Zustand gebracht wird. Alle anderen Indizes sind illegal. Dass ein vernünftiger Index im Bereich von null bis `Argumente()` angegeben wird, muss vom Programmierer sichergestellt werden.

10.2 Implizites „jenach“

× Die Adventure-Bibliothek benutzt häufig eine Fallunterscheidung in Routinen, die nicht ausdrücklich mit `jenach` angegeben ist:

```
vorher = {
    nehmen: Stop "Das geht nicht!";
    untersuchen:
    sonst: Stop Groß(der, selbst), " ", vist(),
        " nicht wichtig.";
}
```

Dies wird mit einem schmutzigen Trick gemacht: ZETA behandelt alle Bereichsangaben vor einem Doppelpunkt, die auf der untersten Ebene der Routine stehen, als den Beginn einer `jenach`-Anweisung, die diese Bereiche gegen den wert der globalen Variable mit dem Index 255 überprüft. ZETA reserviert die Variablen 252 bis 255 als Register für den internen Gebrauch. Die Routine oben sieht in Wirklichkeit also so aus:

```
vorher = {
    jenach (leseArg(255)) {
        nehmen: Stop "Das geht nicht!";
        untersuchen:
        sonst: Stop Groß(der, selbst), " ", vist(),
            " nicht wichtig.";
    }
}
```

und die Bibliothek ruft sie folgendermaßen auf:

```
schreibeArg(255, aBefehl);
aObj.vorher();
```

Auf diese Weise kann der Quelltext eines Adventures übersichtlich gestaltet werden. Diese Vorgehensweise ist jedoch mit Vorsicht zu genießen. Direkte Zugriffe auf Variablen bergen ein hohes Risiko. Sie werden trotzdem erlaubt, da sie erfahrenen Programmieren auch neue Möglichkeiten eröffnen.

11 Daten für den Parser

11.1 Definition von Objektvokabular

Die Eigenschaften `Subst` und `Adj` zum Definieren von Objektvokabular sind keine gewöhnlichen Eigenschaften. Sie verdienen daher eine Erklärung.

Zunächst sind `Subst` und `Adj` *keine* Eigenschaften im Sinne einer Objekteigenschaft. Sie sind vielmehr Schlüsselwörter, um das Vokabular, das der Parser für dieses Objekt versteht, zu definieren. Ein Ausdruck wie `Hufeisen.Subst[0]` ist daher nicht gültig und erzeugt eine Fehlermeldung.

Die Syntax für die Angaben von Vokabeln bei Objekten ist ebenfalls von der für andere Eigenschaften verschieden. Es muss immer eine Vokabel in Hochkommas angegeben werden, gefolgt von einem konstanten Wert, der diese Vokabel näher beschreibt – bei Substantiven ist dies meist der Genus. Diese Angaben stehen immer direkt zusammen, Kommas trennen sie von weiteren Angaben:

```
Subst = 'Substantiv' {Konstante}, ...


Adj = 'Adjektiv' {Konstante}, ...
```

Diese Definition wird von ZETA umgewandelt, so dass sie drei andere Eigenschaften, die dem Autoren auf den ersten Blick nicht ersichtlich sind, belegen. Diese Eigenschaften sind für jedes Objekt reserviert und werden wie folgt belegt:

`__vok1`	erster Teil der Vokabel
`__vok2`	zweiter Teil der Vokabel
`__vokf`	Flaggen für die Vokabel

ZETA teilt die Vokabel in zwei Teile. Jede Vokabel hat im *Z*-Format die Auflösung von neun Zeichen, Sonderzeichen, die keine Buchstaben sind (wie Ziffern oder Satzzeichen) belegen zwei Zwichen, genauso wie die Umlaute und das scharfe s, die in `ae`, `oe`, `ue`und `ss` umgewandlt werden. Folgende Wörter sind für die *Z*-Maschine gleich:

```
'schwarzbraun'      ->   'schwarzbr'
'schwarzbrot'       ->   'schwarzbr'
'schwarzbrenner'    ->   'schwarzbr'
```

Für Verben, Präpositionen und Artikel ist diese Auflösung (in den meisten Fällen) ausreichend, bei Substantiven, die häufig zusammengezsetzt sind, und Adjektiven stößt man hier schnell an die Grenzen. Da das Objektvokabular auch zur Unterscheidung mehrerer Objekte herangezogen wird, besteht ein Substantiv oder Adjektiv in Wirklichkeit aus zwei *Z*-Vokabeln. Die zweite Vokabel kann den Wert null haben, was bedeutet, dass das Wort kurz genug ist, um in eine Vokabel zu passen.

Die Flaggen auf `__vokf` sind die zur Vokabel angegebenen Konstanten plus einer Kennzeichnung: `$8000` für Substantive, `$4000` für Adjektive.

Ein Objekt mit der Definition

```
Adj = 'neu';
Subst = 'geldschein' m, '20-euro-schein' m, 'banknote' f;
```

erhält also in Wirklichkeit folgende Eigenschaften:

```
__vok1 = 'neu',  'geldschei',  '20-eur',    'banknote';
__vok2 = 0,      'n\',         'o-schein',  0;
__vokf = $4000,  $8000 + m,    $8000 + m,   $8000 + f
```

Der 20-Euro-Schein nutzt sogar die gesamte Auflösung von 18 Zeichen aus, da die vier Sonderzeichen doppelt zählen.

11.2 Grammatikdefinitionen

Das Format der Grammatikdefinitionen ist nicht, wie häufig vermutet wird, von der *Z*-Maschine vorgegeben, sondern wird von ZETA festgelegt. Die folgenden Direktiven erzeugen die für den Parser benötigten Daten:

```
Befehl            Präposition       Synonym
Befehlseintrag    Satzbaustein      Verb
```

Nachfolgend wird die Struktur der Grammatikdaten beschrieben. Diese Information ist nur interessant, wenn man die Bibliothek verändern will oder wenn man eine eigenen Parser mit ZETA entwickeln will.

1 Allgemeines

Die Grammatikdefinitionen beginnen am unteren Ende des statischen Speicherbereichs. Die Adresse, in der sich der statische Speicherbereich befindet, steht als 2-Bit-Zahl in Bytes 14 und 15 im Dateikopf (`0-->7`).

Die ersten vier Words der Grammatikdefinitionen enthalten die Adressen der Unterlisten:

`$00`	Befehlsliste
`$01`	Verbliste
`$02`	Synonymliste
`$03`	Präpositionsliste

2 Befehle

Befehle sind Zusammenschlüsse von Daten, die in Textadventures dazu benutzt werden, Routinen und Variablen für die Ausführung der vom Spieler eingegebenen Befehle zu gruppieren, um sie einer Eingabesyntax zuordnen zu können. Sie werden mit der Direktive

`Befehl` *Bezeichner* `{ ... }`

definiert, innerhalb der geschweiften Klammern stehen so genannte *Befehlseinträge*, die ähnlich wie Eigenschaften von Objekten angegeben werden. Bevor man einen Befehl definieren kann, müssen diese Einträge mit

`Befehlseintrag` *Eintrag*`;`

deklariert werden. Dies passiert normalerweise in der Bibliothek und der Autor eines Spiels muss sich keine Gedanken darüber machen.

Das Konzept des Befehls in ZETA ist das ein stark vereinfachtes Objekts: Ein Befehl hat Einträge, so wie ein Objekt Eigenschaften hat. Diese Einträge können aber nur je einen Wert enthalten, Listen sind nicht möglich. Ausserdem hat ein Befehl keine Position im Objektbaum und keine Attribute.

Das erste Wort in der Befehlsliste gibt die Anzahl der Befehle an. Es folgt ein Wort pro Befehl, das die Adresse seiner Eintragsliste enthält.

Die Eintragsliste eines Befehls sieht folgendermaßen aus: Sie beginnt mit einem Byte, das eine Null enthält. Danach folgen die angegebenen Einträge, und zwar in absteigender Reihenfolge ihrer internen IDs. Pro Eintrag werden drei Bytes angegeben: Das erste enthält die ID des

Eintrags plus 64. Die nächsten beiden Bytes enthalten den Wert des Eintrags. Diese Liste wird abgeschlossen, wenn das erste dieser Bytes null ist.

Dieses Format sieht etwas ungewöhnlich aus. Wozu ist die Null am Anfang und das Addieren von 64 notwendig? Dieses Format ist das Format der Eigenschaften von Objekten in der *Z*-Maschine. Da Befehle keine Objekte sind – weder im Sinne von ZETA noch im Sinne der *Z*-Maschine – kann man auf die Einträge nicht mit der Notation nicht wie auf Eigenschaften von Objekte zugreifen und ihnen auch keine Mitteilungen senden.

Mit einem schmutzigen Trick geht dies jedoch. Der Trick soll hier verschwiegen werden, er wird in der Systemfunktion `leseEintrag(b, e)` angewandt, die den Eintrag `e` eines Befehls`b` zurückgibt.

3 Verben

Die Verbliste enthält alle als Verb definierten Vokabeln sowie die Satzmuster für jedes Verb. Die Definition von Verben wird in Abschnitt 9.2 beschrieben.

Das erste Wort der Verbliste enthält die Anzahl der Verben. Für jedes Verb wird nun ein Block aus drei Wörtern angegeben, der die regelmässige Form, die unregelmässige Form (oder null) und die Adresse des Satzmusters enthält. Verben, die eine regelmäßige und eine unregelmäßige Form haben, werden mit einem Gleichzeichen definiert:

```
Verb 'trete' = 'tritt', 'stosse', ...
```

Hier würde `'trete'` im ersten Wort, `'tritt'` im zweiten abgelegt. `'stosse'` ist ein neues Verb, das aber auf dasselbe Satzmuster verweist.

Satzmuster sind Ketten von beliebig vielen Satzbausteinen, die jeweils einem Befehl zugeordnet werden. Jedes Satzmuster beginnt mit einem Wort, das den Befehl enthält oder eine Null, um das Ende der Liste zu markieren. Danach folgen die Satzbausteine, die je drei Bytes belegen. Das erste Byte gibt den Typ des Satzbausteins an, die letzten Bytes enthalten ein zum Satzbaustein gehörendes Datenwort. Ist das erste Byte null, so wird das Satzmuster abgeschlossen. Es gibt folgende Typen von Satzbausteinen:

`0`	Endmarker
`1`	direktes Objekt (`sb`)
`2`	indirektes Objekt (`sb + p`)
`3`	Präposition
`4`	ein explizit angegebenes Wort
`5`	eine Routine

Der Inhalt des Datenworts ist eine Vokabeladresse (3 und 4), die Adresse einer Routine (5), oder ein vorher definierter Satzbaustein, im Fall des indirekten Objekts enthält das erste Byte den Satzbaustein, das zweite die Präposition. Satzbausteine müssen deklariert werden, bevor sie in Satzmustern verwendet werden:

```
Satzbaustein dasObjekt;
```

Die internen Nummern von Satzbausteinen sind immer Vielfache von 256, so dass `sb + p` immer eindeutig ist.

Folgende Definition

```
Verb 'ziehe'
  * dasObjekt       -> ziehen;
  * dasObjekt 'an'; -> anziehen;
```

erzeugt diese Liste von Satzmustern:

00	ziehen	Befehl (Wort)
02	1	direktes Objekt
03	dasObjekt	Satzbaustein (Wort)
05	0	Ende des Satzmusters
06	anziehen	Befehl (Wort)
08	1	direktes Objekt
09	dasObjekt	Satzbaustein (Wort)
11	3	Präposition
12	'an'	Vokabel (Wort)
14	0	Ende des Satzmusters
15	0	Ende der Satzmusterliste (Wort)
17	...	Nächste Satzmusterliste

Die Trennung der Typen 3 und 4 erscheint etwas willkürlich, ist aber für den zukünftigen Gebrauch gemacht worden, um etwa Präpositionen als Teile von Verben zu betrachten.

4 Präpositionen

Das erste Wort der Präpositionsliste gibt die Anzahl der Präpositionen an. Es folgt zu jeder Präposition ein Block mit vier Wörtern, der die Vokabeladressen dr Präposition und von bis zu drei Pronomialadverbien (wie „damit“ oder „darauf“). Wenn eine Präposition weniger als vier Einträge besizt, so wird mit Nullen aufgefüllt. Zum Beispiel bewirkt die Direktive

```
Präposition 'mit' -> 'damit';
```

den Präpositionsblock:

```
'mit', 'damit', 0, 0
```

5 Synonyme

Synonyme werden hintereinader auf eine Liste geschrieben. Jedes Synonym eresetzt `m` Wörter durch `n` andere Wörter, wobei `n` auch null sein kann. Das erste Byte jedes Synonyms enthält die Anzahl der Wörter in der Form `m*16 + n`. Es folgen `n+m` Wörter, die die Vokabeladressen enthalten. Zum Beispiel erzeugt:

```
Synonym 'ins' -> 'in' 'das';
```

den Synonymeintrag

```
$12, 'ins', 'in', 'das';
```

Die Synonymliste endet, wenn das Byte, das die Anzahl der Wörter festlegt, null ist.

Anhang A:

Spezifikation der Sprache

ZETA ist ein Compiler für die virtuelle *Z*-Maschine. Es können Spieldateien der Versionen fünf und acht erzeugt werden. Für diese Art von Spieldateien steht eine breite Palette an Interpretern für verschiedene Plattformen zur Verfügung.

Diese Spezifikation ist eine sehr trockene, aber komplette Auflistung aller Sprachelemente von ZETA. Wer bereits Autorensysteme oder Programmiersprachen kannt, kann hier einen schnellen Überblick über ZETA bekommen. Ein behutsamerer Einstieg in die Sprache von Zeta findet sich in [nicht gefunden].

Folgende typographische Konventionen gelten: *Platzhalter* in Kursiv müssen durch passende Ausdrücke ersetzt werden. Optionale Angaben sind in {geschweifte Klammern} gesetzt. Wörtlicher Text, also Schlüsselwörter und Operatoren, sind mit `Schreibmaschine` geschrieben.

Format des Quelltexts

Der Quelltext ist ASCII-Text, der mit dem Zeichensatz Zeichensatz ISO-8859-1 (Westeuropa) gelesen wird. Er besteht aus *white space* (einer beliebig langen Folge von Leerzeilen, Tabulatoren und Zeilenumbrüchen), aus Bezeichnern, numerischen Konstanten, Operatoren und Texten, die in doppelte oder einfache Anführungszeichen eingeschlossen werden. Bezeichner müssen durch *white space* voneinander abgetrennt werden, Operatoren und Bezeichner können direkt aufeinander folgen. Syntaktische Zeilen können über mehrere Textzeilen gehen und müssen mit einem Semikolon abgeschlossen werden. Die geschweiften Klammern werden stets als eigene Zeile betrachtet. Außerhalb von Anführungszeichen werden Buchstaben mit Akzenten in ihre lateinischen Äquivalente umgewandelt.

Kommentare

Alles, was auf einer Textzeile nach zwei aufeinanderfolgenden Schrägstrichen (`//`) steht, wird vom Compiler ignoriert und als Kommentar betrachtet. Ebenso ist alles, was auf beliebig vielen Zeilen zwischen der Anfangsmarke `/*` und der Endmarke `*/` steht, ein Kommentar.

Bezeichner

Bezeichner können bis zu 32 Zeichen lang sein und lateinische Buchstaben, Ziffern und den Unterstrich enthalten. Groß- und Kleinschreibung wird nicht unterschieden. Bezeichner können die deutschen Umlaute und das scharfe s enthalten, diese werde jedoch als äquivalent zu `ae`, `oe`, `ue` und `ss` betrachtet. Das erste Zeichen eines Bezeichners darf keine Ziffer sein.

Bezeichner können lokal oder global sein. Die meisten Bezeichner, etwa für Routinen, Eigenschaften oder Objekte, sind global. Alle globalen Bezeichner müssen voneinander verschieden sein. Dazu zählen die Bezeichner der Systemfunktionen und die Schlüsselwörter von ZETA. Alles, was innerhalb von Routinen oder Anweisungsblöcken mit geschweiften Klammern definiert wird, ist lokal. Alle lokalen Bezeichner müssen voneinander verschieden sein. Lokale und globale Bezeichner dürfen dieselben Namen haben. Im Gültigkeitsbereich der lokalen Bezeichner sind dann die entsprechenden globalen Bezeichner nicht zugänglich.

Datentypen

Zahlen

Zahlen sind Folgen von Ziffern. Normalerweise werden Zahlen als ganze Dezimalzahlen betrachtet, die Zahlen in ZETA können keine Brüche oder Nachkommastellen haben. Zahlen in ZETA können Werte von –32768 bis 32767 annehmen.

Zahlen können in hexadezimaler Form angegeben werden, wenn man ihnen ein Dollarzeichen voranstellt. Die „Ziffern“ 10 bis 15 werden dann durch die Buchstaben A bis F dargestellt. Binärzahlen können nach zwei einleitenden Dollarzeichen angegeben werden, sie dürfen nur die Ziffern 0 und 1 enthalten. Binär- und Hexadezimalzahlen können im Bereich von `$0000` bis `$FFFF` liegen.

Routinen

Routinen sind Folgen von Anweisungen, die in einem Block zwischen geschweiften Klammern zusammengefasst sind. Diese Anweisungen können ausgeführt werden, wenn die Routine aufgerufen wird. Beim Aufruf kann die Routine Argumente erhalten und einen Wert zurückgeben.

ZSCII-Zeichen

Ein ZSCII-Zeichen ist ein gültiges Zeichen der *Z*-Maschine. Dies sind die Zeichen des 7-Bit-ASCII-Zeichensatz plus einige Sonderzeichen. Ein ZSCII-Zeichen wird als einzelner Buchstabe zwischen Hochkommas angegeben (`'x'`).

Vokabeln

Vokabeln sind kurze Texte in einfachen Anführungszeichen, die im internen Wörterbuch einsortiert werden. Vokabeln können nur Kleinbuchstaben, Ziffern und Satzzeichen enthalten. Großbuchstaben und Umlaute werden umgewandelt. Eine Vokabel kann höchstens neun Zeichen lang sein. Eine Vokabel, die nur einen Buchstaben enthält, muss als letztes Zeichen einen rückläufigen Schrägstrich, den Backslash, angehängt bekommen, um sie von ZSCII-Zeichen zu unterscheiden. Vokabeln dienen im Allgemeinen der Texteingabe.

Texte

Texte sind Zeichenketten, die zwischen doppelten Anführungszeichen (Gänsefüßchen) stehen und die zur Textausgabe gedacht sind. Texte können über mehrere Zeilen gehen. Innerhalb von Texten werden Zeilenumbrüche wie ein einzelnes Leerzeichen betrachtet.

Leerzeilen in Texten werden als Absatz interpretiert. Weiterhin gibt es folgende Sonderzeichen:

`\n`	Zeilenumbruch (newline)
`\p`	Absatz (paragraph)
`""`	geschütztes Anführungszeichen
`\"`	geschütztes Anführungszeichen (alternativ)
`\{`	geschützte Klammer auf
`\}`	geschützte Klammer zu
`\`	geschütztes Leerzeichen
`\\`	Backslash

Weiterhin gibt es Sequenzen von Sonderzeichen, die Zeichen mit Akzenten erzeugen, siehe Anhang. Texte können auch Kurzstrings in geschweiften Klammern enthalten, die mit der Direktive

`Kurzstring` *Name* `"`*String*`"`

deklariert werden und mit derselben Anweisung innerhalb von Routinen geändert werden können.

Datenstrukturen

Konstanten

Konstanten sind Bezeichner für Werte, die sich nicht ändern können. Es können beliebig viele

Konstanten definiert werden mit

```
Konstante Name {, Name ...} {= Wert};
```

Als Wert kann ein beliebiger Datentyp angegeben werden, auch ein Ausdruck, wenn dieser konstant ist. Wird kein Wert angegeben, so bekommt die Konstante einen Wert, der von allen anderen Konstanten, die ohne Wert definiert wurden, verschieden ist.

Die Konstanten `falsch`, `nichts` und `null` mit dem Wert null und `wahr` mit dem Wert eins sind bereits in ZETA vordefiniert.

Globale Variablen

Globale Variablen sind Speicherplätze für Werte, die von überall im Programm aus zugänglich sind. Sie können beliebige Datentypen enthalten. Es können bis zu 234 globale Variablen definiert werden mit

```
Variable Name {, Name ...} {= Wert};
```

Wert ist hier eine Konstante, die den Anfangswert er Variablen angibt. Wird kein *Wert* angegeben, so ist der Anfangswert null.

Objekte

Objekte sind Zusammenschlüsse von Werten, den so genannten Eigenschaften. Objekte in ZETA haben außerdem eine Position in einem Objektbaum und können Attribute haben. Sie werden weiter unten genauer beschrieben.

Felder

Felder sind Datenstrukturen, die eine bestimmte Anzahl von l von einfachen Datenelementen enthalten können. Die Anzahl der Feldeinträge ist konstant, die Einträge können sich ändern. Ein Feld wird folgendermaßen deklariert:

```
Feld [Dimension];
```

um ein Feld mit *Dimension* Einträgen zu erzeugen, die zu Beginn des Spiels alle null sind, oder

```
Feld = Werteliste;
```

um ein Feld mit den Einträgen der *Werteliste* zu erzeugen oder

```
Feld = "Zeichenfolge";
```

um ein Feld von ZSCII-Zeichen zu erzeugen, dessen Einträge die Zeichen der angegebenen Zeichenfolge sind.

Strings sind Felder, deren Einträge Bytes sind, die nur Werte von null bis 255 annehmen können. Sie werden analog zu Feldern mit dem Schlüsselwort `String` deklariert:

```
String [Dimension];
String = Werteliste;
String = "Zeichenfolge";
```

Anweisungen

lokale Variablen

Lokale Variablen werden wie folgt deklariert:

```
lokal Variablenliste;
```

Eine *Variablenliste* hat die Form:

Bezeichner {= *Ausdruck*} {, *Variablenliste*}

Die Anweisung lokal muss stets die erste in einem Anweisungsblock sein oder einer anderen lokal-Anweisung folgen. Die Gültigkeit der lokalen Variable beschränkt sich auf den Anweisungsblock, in dem sie deklariert wurde. Es können maximal fünfzehn lokale Varaiblen definiert werden

Ausdrücke

Ausdrücke werden in ZETA in algebraischer Notation angegeben. Ein Ausdruck hat die Form

{*linker Operand*} *Operator* {*rechter Operand*}

Die Operanden können Datentypen oder wiederum Ausdrücke sein. Nicht jeder Operator verlangt nach einem linken oder rechten Operanden. Operatoren, die beide Operanden verlangen, heißen binär, Operatoren, die nur auf einen Operanden wirken, heißen unär.

Es folgt eine Liste der Operatoren in aufsteigender Reihenfolge ihrer Präzedenz:

`a, b`	verkettet zwei Ausdrücke
`a = b`	weist `a` den Wert von `b` zu
`a || b`	wahr, wenn `a` oder `b` wahr ist
`a oder b`	alternative Schreibweise von `a || b`
`a && b`	wahr, wenn `a` und `b` wahr sind
`a und b`	alternative Schreibweise von `a && b`
`a == b`	wahr, wenn `a` gleich `b` ist
`a != b`	wahr, wenn `a` ungleich `b` ist
`a > b`	wahr, wenn `a` grösser als `b` ist
`a < b`	wahr, wenn `a` kleiner als `b` ist
`a >= b`	wahr, wenn `a` grösser als oder gleich `b` ist
`a <= b`	wahr, wenn `a` kleiner als oder gleich `b` ist
`o ist a`	wahr, wenn `o` das Attribut `a` besitzt
`o hat p`	wahr, wenn `o` die Eigenschaft `p` definiert hat
`a in b`	wahr, wenn Objekt `a` ein Kind von Objekt `b` ist
`o beerbt k`	wahr, wenn `o` eine Instanz der Klasse `k` ist
`a + b`	addiert `a`und `b`
`a - b`	subtrahiert `a`und `b`
`a * b`	multipliziert `a`mit `b`
`a / b`	dividiert `a`durch `b` (ganzzahlig)
`a % b`	Rest der Ganzzahlendivision von `a` durch `b`
`a & b`	bitweise Und-Verknüpfung von `a` und `b`
`a | b`	bitweise Oder-Verknüpfung von `a` und `b`
`a ^ b`	bitweise Exklusiv-Oder-Verknüpfung von `a`und `b`
`-a`	arithmetische Negation von `a`
`!a`	logische Negation von `a`
`nicht a`	alternative Schreibweise von `!a`)
`~a`	bitweise Umkehrung von `a` (Einerkomplement)
`a<<b`	Bit-Verschiebung in `a` um `b` Stellen nach links
`a>>b`	Bit-Verschiebung in `a` um `b` Stellen nach rechts
`a-->b`	Eintrag `b` aus `a` als Wort lesen
`a->b`	Eintrag `b` aus `a` als Byte lesen
`a++`	`a` lesen, dann um eins erhöhen

`a--`	`a` lesen, dann um eins verringern
`++a`	`a` um eins erhöhen, dann lesen
`--a`	`a` um eins verringern, dann lesen
`o.p`	Eigenschaft `p` von Objekt `o`
`o.#p`	Anzahl der Einträge von `p` in `o`
`o.&p`	Feldadresse der Eigenschaft `p` von `o`
`a[i]`	`i`-ten Eintrag des Feldes `a`lesen.
`a(...)`	Auswerten von `a`, eventuell mit Argumenten
`a(...)`	Aufruf der Funktion `a`, eventuell mit Argumenten

Operatoren mit höherer Präzedenz binden ihre Operanden stärker, so dass `a*b + c` als `(a*b) + c` interpretiert wird. Die Präzedenz kann durch die Angabe von runden Klammern verändert werden, zum Beispiel `a*(b + c)`.

Alle arithmetischen Operationen und Vergleiche werden vorzeichenbehaftet durchgeführt. Logische Operatoren haben als Ergebnis stets einen der Werte `wahr` (eins) oder `falsch` (null). Bitweise Operatoren betrachten die Werte als 16-Bit-Werte ohne Vorzeichen.

Einige Operatoren vereinfachen die Zuweisung, ihre Präzedenz ist wie die der einfachen Zuweisung `=`:

`a += b`	entspricht	`a = a + b`
`a -= b`	entspricht	`a = a - b`
`a *= b`	entspricht	`a = a * b`
`a /= b`	entspricht	`a = a / b`
`a %= b`	entspricht	`a = a % b`
`a &= b`	entspricht	`a = a & b`
`a |= b`	entspricht	`a = a | b`
`a <<= b`	entspricht	`a = a << b`
`a >>= b`	entspricht	`a = a >> b`

Zuweisungen sind Ausdrücke, die als Ergebnis den Wert der linken Seite nach der Zuweisung haben: `a = b = 5` weist `a` den Wert von `(b = c)` zu, der 5 ist. Nach dieser Anweisung ist der Wert von `a` und `b` fünf. `a = b += 5` erhöht `b` um fünf und weist `a` den neuen Wert von `b` zu. Meistens werden Zuweisungen aber als Anweisung eingesetzt, ohne dass dabei ihr Ergebnis berücksichtigt wird.

Ausdrücke, die ausgewertet werden, aber das Ergebnis nicht verwenden, sind nicht erlaubt und werden nicht kompiliert. Die einzigen Ausdruecke, die alleine stehen koennen sind Zuweisungen, Inkremente und Dekremente.

Ausnahme ist hier ein Text in Anführungszeichen oder eine Konstante, die einen Text enthält: Diese Ausdrücke geben diesen Text aus, wenn sie ausgewertet werden. Das passiert nur, wenn sie alleine stehen, Folgende Anweisung gibt keinen Text aus:

```
x = "Vincent van Gogh";
```

Das Komma als Operator zum Verketten von zwei Ausdrücken ist verschieden vom Komma, das zum Abtrennen von Argumenten in einer Funktion verwendet wird. Beim Verketten von Ausdrücken werden alle Ausdrücke von links nach rechts ausgewertet, der Wert der Verkettung ist der Wert des rechten Ausdrucks.

Diese Verkettung ist vor allem in Schleifen oder Anweisungen zur Ausgabe von Text sinnvoll:

```
schleife (i=0, j=10; j; i++, j--) ...
"Du hast ", num(selbst.Anzahl), " Murmeln.";
```

Funktionsaufrufe

Ein Funktionsaufruf ist hat folgende Form:

Routine ({*Argumente*})

Routine ist der Bezeichner einer woanders im Quelltext definierten globalen Routine. Die *Argumente* können fehlen, oder eine Anzahl von Werten sein, die durch Kommas getrennt sind und die an die Routine übergeben werden. Auch, wenn die Argumente fehlen, muss die Klammer angegeben werden.

Beim Aufruf der Routine werden zunächst die lokalen Variablen auf die Werte der übergebenen Argumente gesetzt, dann werden die Anweisungen in der Routine abgearbeitet. Der Wert des Aufrufs ist der Rückgabewert der Routine.

Globale Routinen werden mit

`Routine` *Name* {(*Argumentliste*)} `{` *Anweisungen* `}`

definiert. Routinen müssen nicht unbedingt vor ihrem Aufruf definiert werden.

Auswerten von Variablen

Eine Variable kann mit einer ähnlichen Syntax ausgewertet werden:

Variable(*Argumente*)

Die *Argumente* sind wieder optional. Wenn die Variable auf eine Routine verweist, wird diese mit den Argumenten aufgerufen, das Ergenbis der Auswertung ist der Rückgabewert der Routine. Wenn die Variable einen Text enthält, so wird dieser ausgegeben, das Ergebnis ist wahr. Ansonsten passiert nichts, das Ergebnis ist der Wert der Variablen.

Rückgabe von Werten

Eine Routine kann einen Wert an die aufrufende Funktion zurückgeben, die Syntax dazu ist:

```
antworte {Ausdruck};
```

Die Routine wird sofort verlassen, der Wert des Ausdrucks wird an die aufrufende Routine zurückgegeben. Fehlt der Ausdruck, so wird `falsch` zurückgegeben.

Die Anweisung `Stop` gibt den Wert `wahr` zurück. `Stop` kann auch in der Form

```
Stop Ausdrücke
```

angegeben werden, dann werden erst die Ausdrücke ausgewertet (meist Texte), bevor die Routine mit dem Wert `wahr` verlassen wird. Eine Routine, die keine der Anweisungen `Stop` oder `antworte` enthält, gibt bei nach dem Ausführen aller Anweisungen `falsch` oder null zurück.

Bedingungen

Manche Anweisungen werden nur unter bestimmten Bedingungen ausgeführt. Die allgemeine Form einer bedingten Anweisung ist:

```
wenn (Bedingung) Anweisung
{sonst Anweisung}
```

Anstelle der *Anweisung* kann eine einzelne Anweisung, die mit einem Semikolon abgeschlossen wird, oder eine Reihe beliebig vieler Anweisungen, die mit `{` und `}` gruppiert werden, stehen.

Eine *Bedingung* ist ein Wahrheitswert. Der Wert `falsch` bedeutet, dass die Bedingung falsch ist, alle anderen Werte bedeuten, dass die Bedingung wahr ist.

Die Angabe einer zweiten Anweisung nach `sonst` ist optional. `sonst` bezieht sich immer auf das zuletzt definierte `wenn`, wenn die Zugehörigkeit nicht durch geschweifte Klammern eindeutig ist.

Eine Kurzform von `wenn`, in der lange wenn-dann-Konstrukte eine Variable gegen mehrere konstante Werte prüfen, ist `jenach`:

```
jenach (Ausdruck) {
    {Bereiche}
    {sonst: Anweisungen}
}
```

Bereiche haben hier die Form

Bereich`:` *Anweisungen*

Ein *Bereich* sind konstante Werte, die durch Kommas getrennt sind oder Bereiche von konstanten Werten in der Form *von*`...`*bis*. Der spezielle Bereich `sonst` deckt alle bisher nicht behandelten Fälle ab und muss als letzter stehen.

Die *Anweisungen* sind eine beliebige Anzahl von Anweisungen, auch keine, die *nicht* mit geschweiften Klammern gruppiert werden. Die Ausführung der Anweisungen wird mit der nächsten Bereichsdefinition beendet und nach der letzten geschweiften Klammer fortgesetzt.

Schleifen

Die `solange`-Schleife hat die Form

`solange (`*Bedingung*`)` *Anweisung*

Für *Bedingung* und *Anweisung* gilt das bei `wenn` Gesagte. Die Anweisung wird solange wiederholt, bis die Bedingung falsch ist. Die Bedingung wird vor Ausführung der Anweisungen überprüft, so dass die Anweisungen nicht ausgeführt werden, wenn die Bedingung bei der ersten Überprüfung falsch ist.

Die `wiederhole-bis`-Schleife hat die Form

```
wiederhole Anweisung
bis (Bedingung)
```

Hier werden die Anweisungen wiederholt, bis die Bedingung wahr ist. Die Überprüfung findet nach dem Ausführen der Anweisungen statt, so dass die Anweisungen mindestens einmal ausgeführt werden.

Eine allgemeine Form der Schleife ist `Schleife`:

`Schleife (`*Init*`;` *Bedingung*`;` *Aktualisierung*`)` *Anweisung*

`Schleife` ist eine Abkürzung für

```
Init
solange (Bedingung) {
    Anweisung
    Aktualisierung
}
```

Jede der drei Angaben in der Klammer kann weggelassen werden.

Mit der Anweisung `abbrechen` springt man aus einer Schleife heraus, mit `weiter` springt man zum nächsten Durchgang der Schleife, wenn dieser noch durchgeführt werden soll. Weiter springt zur nächsten Überprüfung der *Bedingung*, bei Schleife zur *Aktualisierung*.

Sprunganweisungen

Mit der Anweisung

`springe` *Sprungmarke*`;`

kann man an eine Stelle in der momentanen Routine springen. Diese Stelle muss mit

`.`*Sprungmarke*`;`

markiert werden. Die Sprungmarke ist nur innerhalb der Routine zugänglich, man kann nicht aus einer Routine herausspringen.

Objekte

Objektbaum

Jedes Objekt hat eine Lage im Objektbaum, diese wird durch seine Mutter, seine Schwester und durch seine Tochter gekennzeichnet, die alle null sein können. Ein Objekt, das keine Mutter hat, hat auch keine Geschwister.

Die Definition eines Objekts sieht folgendermaßen aus:

`Objekt` {*Pfeile*} {*Name*} {`in` *Objekt*} {`{` *Objektkörper* `}`}

Dabei ist jede der Angaben optional. *Pfeile* ist eine beliebige Folge von Pfeilen (`->`), die die anfängliche Lage des Objekts im Baum darstellen. Ein Objekt, das Pfeile hat, ist ein Kind des

zuletzt mit einem Pfeil weniger definierten Objekts. Eine alternative Möglichkeit, die Lage im Baum anzugeben ist, das Mutterobjekt direkt nach `in` anzugeben.

Objekte werden von zwei ab laufend nummeriert. Auf diese Weise sind die Konstanten `falsch`, `wahr` und alle Objektbezeichner verschieden.

Eigenschaften und Attribute

Der Objektkörper kann folgende (optionalen) Angaben enthalten:

> `beerbt` *Klassenliste*`;`
> `ist` *Attributliste*`;`
> *Eigenschaft* {`=` *Werteliste*}`;`

Es können beliebig viele Eigenschaften angegeben werden, zu jeder Eigenschaft kann eine Liste mit einem oder meheren Einträgen angegeben werden.

Formal gibt es zwei Arten von Eigenschaften. Gemeinsame Eigenschaften sind Eigenschaften, die dem Format der Objekte in der *Z*-Maschine entsprechen. Sie müssen mit

> `Eigenschaft` {`+`} *Name* {`oder` *Name*} {`=` *Wert*}`;`

definiert werden und können einen Default-Wert besitzen, der Objekten zugeordent ist, die diese Eigenschaft nicht explizit definiert haben. Es gibt maximal 63 gemeinsame Eigenschaften. Die *Werteliste* kann bei einer gemeinsamen Eigenschaft je Objekt bis zu 32 Einträge haben.

Individuelle Eigenschaften sind eine zusätzliche Datenstruktur von ZETA. Diese Eigenschaften müssen nicht vorher definiert werden und sind in ihrer Anzahl unbegrenzt. Eine individuelle Eigeschaft kann je Objekt bis zu 128 Einträge enthalten. Wenn eine individuelle Eigenschaft von einem Objekt gelesen wird, das sie nicht besitzt, so ist der Wert null.

Einträge von Eigenschaften können jeden Datentyp enthalten, in Listen auch gemischt. Als Wert einer Eigenschaft kann auch eine Routine in geschweiften Klammern angegeben werden, dann können keine weiteren Einträge folgen. (Obwohl die Eigenschaft durch Vererbung weitere Einträge erhalten kann.)

Die *Attributliste* besteht aus durch Kommas getrennte Attributen, die durch ein vorangestelltes `nicht` oder `!` verneint werden können. Bevor ein Attribut angegeben werden kann, muss es mit

> `Attribut` *Name* {`oder` *Name*}`;`

definiert werden. Ein Attribut ist eine vereinfachte Eigenschaft. Ein Objekt hat ein Attribut oder hat es nicht. Es können bis zu 48 Attribute definiert werden.

Mitteilungen

Objekte können Mitteilungen erhalten, die die Form

{*Antwort* `=`} *Objekt*`.`*Eigenschaft* `(`{*Argumente*}`);`

haben. Die runden Klammern sind wie beim Funktionsaufruf zwingend. Je nach Datentyp der Eigenschaft passiert folgendes: Eine Routine wird mit den *Argumenten* aufgerufen, die *Antwort* ist der Rückgabewert der Routine. Ein Text wird ausgegeben, die *Antwort* ist wahr. Alle anderen Datentypen bewirken nichts und antworten mit ihrem Wert.

Spezielle Anweisungen für Objekte

Mit der Anweisung

`verschiebe` *Objekt1* `nach` *Objekt2*`;`

kann *Objekt1* im Objektbaum nach *Objekt2* verschoben werden, *Objekt1* ist dann die älteste Tochter von *Objekt2*. Mit

`entferne` *Objekt*`;`

kann man ein Objekt aus dem Baum entfernen, so dass es die Mutter `nichts` hat.

Attribute in einem Objekt werden mit

`mache` *Objekt Attributliste*`;`

gesetzt, wobei beliebig viele Attribute angegeben werden können. Die Attribute können auch durch ein `!` oder `nicht` negiert werden, dann werden sie gelöscht.

Die Schleife

`alleObjekte` *Variable Anweisungen*

durchläuft alle Objekte nacheinander in der Reihenfolge, wie sie im Quelltext definiert wurden.

Klassen

Objekte können Eigenschaften und Attribute von einer Anzahl von Klassen erben, in dem eine Liste von Klassen als *Klassenliste* angegeben wird. Ein Objekt besitzt ein Attribut, wenn es eine der Klassen besitzt. Eigenschaften werden von links nach rechts in der Liste vererbt, das heißt, Eigenschaften von weiter rechts stehenden Klassen ersetzen eventuell bereits von anderen Klassen geerbte Eigenschaften. Die zulätzt angegebene Klasse ist dominant.

Beim Vererben werden alle Eigenschaften ersetzt, es sei denn, es handelt sich um gemeine Eigenschaften, die zuvor mit + als erweiternd definiert wurden.

Klassen müssen definiert werden, bevor sie an andere Objekte vererben können, die Definition sieht folgendermaßen aus:

```
Klasse Name {(Anzahl)} { { Klassenkörper } }
```

Der *Klassenkörper* hat hat dasselbe Format wie ein *Objektkörper*, Klassen können ebenfalls von anderen Klassen erben. Eine *Anzahl* in runden Klammern erzeugt so viele Instanzen der Klasse, die nicht im Spiel sind, aber später mit der Systenfunktion `erzeuge` dynamisch erzeugt werden können.

Klassen können auch Mitteilungen erhalten, wenn sie von einer ihrer Instanzen gesendet werden. Die Variablen `selbst` und `Sender` werden dabei nicht geändert.

Grammatikdefinitionen

Als zusätzliche Datenstrukturen für Textadventures gibt es in ZETA Befehle, Verben, Präpoitionen und Synonyme.

Befehle sind objektähnliche Zusammenschlüsse von Daten und werden wie folgt definiert:

```
Befehl Name { Einträge };
```

Einträge haben die Form

```
Eintrag { = Wert };
```

und müssen vor Benutzung mit

```
Befehlseintrag Name;
```

definiert werden. Verben sind Imperativverben, denen mögliche Satzstrukturen zugeordent sind:

`Verb` *Verb Satzmuster*`;`

Verb hat die Form:

regelm. Verb { `=` *unregelm. Verb*} {`,` *Verb*}

Ein *Satzmuster* hat die Form:

`*` *Satzbausteine* `->` *Befehl*`;`

Satzbausteine ist eine nicht durch Kommas getrennte Auflistung folgender Satzbausteine:

Satzbaustein
Routine
`'`*Vokabel*`'`

Das Satzmuster ist beendet, wenn eine Zeile nicht mit einem Stern beginnt. Satzbausteine müssen vorher mit

`Satzbaustein` *Satzbaustein*`;`

deklariert werden. Synonyme sind Listen von Wörtern, die durch andere ersetzt werden sollen:

`Synonym` *Wortliste* `->` *Wortliste*`;`

Präpositionen werden für indirekte Objekte benötigt und können mit bis zu drei Pronomialadverbien angegeben werden:

`Präposition` *Präp* { `->` *Adverb* {*Adverb*} {*Adverb*} }`;`

Steuerbefehle

ZETA besitzt einige Anweisungen, die den Kompiliervorgang steuern können. Diese Anweisungen beginnen mit einem Doppelkreuz (#) und werden mit dem Zeilenende abgeschlossen. (Es kann aber auch ein Semikolon benutzt werden, solange es auf derselben Zeile wie das Doppelkreuz ist. Das Doppelkreuz muss nicht unbedingt in der ersten Spalte der Zeile stehen.)

`#datei "`*Dateiname*`"`
Bindet die Datei mit dem *Dateinamen* ein. Der *Dateiname* kann einen Pfad enthalten. Wird kein Pfad angegeben, so wird der Pfad genommen, in dem der Hauptquelltext liegt.

Eingebundene Dateien können wiederum Dateien einbinden.

`#optionen` -*Optionen*
Setzt diese Optionen, wenn es nicht bereits zu spät dafür ist.

`#wenn` *Ausdruck*
Kompiliert die nachfolgenden Anweisungen nur, wenn der *Ausdruck* wahr ist.

`#wenn def` *Bezeichner*
Kompiliert die nachfolgenden Anweisungen nur, wenn der Bezeichner bereits (explizit) definiert wurde.

`#wenn ndef` *Bezeichner*
Dito, nur negiert

`#sonst`
Kompiliert die nachfolgenden Anweisungen nur, wenn die vorangehenden Anweisungen wegen `#wenn` nicht kompiliert wurden.

`#ende`
Beedndet die bedingte Kompilierung. `#wenn`-Konstrukte können verschachtelt werden.

`#text` {*Fehlercode*} `"`*Text*`"`
Schreibt den Text als Information heraus oder als ZETA-Fehler mit dem angegebenen Fehlercode, der einer der Bezeichner Warnung, Fehler oder Faal sein muss.

`#alias` *Synonym Bezeichner*
Definiert ein *Synonym* zu einem *Bezeichner*. Auf diese Weise kann man zum Beispiel englische Schlüsselwörter definieren.

`#seriennr` *Seriennummer*
Setzt die Seriennummer der Spieldatei.

`#release` *Release*
Setzt die Release, die Veröffentlichung, der Spieldatei.

Systemfunktionen

Einige Funktionen sind bereits in ZETA vorhanden. Meist sind es Funktionen, die einem bestimmten Opcode der *Z*-Maschine entsprechen. Systemfunktionen werden nur kompiliert, wenn sie im Quelltext tatsächlich benutzt werden. Eine Systemfunktion ist genau wie eine

Routne, nur dass die Anweisungen darin nicht von Programmierer kommen, sondern von ZETA. Systemfunktionen werden – im Gegensatz zu normalen Routinen – beim Aufruf auf die Anzahl der Argumente und auf einen Rückgabewert hin überprüft.

Folgende Systemfunktionen stehen in ZETA zur Verfügung:

`aktivesFenster(f)`
Aktiviert das Fenster `f` zum Schreiben. `f` kann null sein für das untere Textfenster oder eins für das obere Statusfenster.

`Argumente()`
gibt die Anzahl der an die Routine übergebenen Argumente zurück. Dies ist wichtig bei Routinen mit einer beliebigen Anzahl von Parametern.

`Ausgabekanal(k)`
Aktiviert den Kanal `k`, wenn `k` positiv ist, oder deaktiviert den Kanal `-k`, wenn `k` negativ ist, für die Ausgabe. Mehrere Kanäle können aktiv sein. Mögliche Werte für `k` sind: 1 für den Bildschirm, 2 für ein Manuskript und 4 für ein Protokoll der eingegebenen Befehle.

`bsBreite()`
gibt die Breite des Bildschirms in Zeichen an.

`bsHöhe`
gibt die Höhe des Bildschirms in Zeichen an.

`Char(z)`
gibt das Zeichen (engl. *character*) mit dem ZSCII-Wert `z` aus.

`CursorPos(x, y)`
positioniert den Cursor an Spalte `x` und Zeile `y` im oberen Fenster. Die linke obere Ecke ist `(1, 1)`.

`CursorX()`
gibt die momentane Spalte der Cursorposition an.

`CursorY()`
gibt die momentane Zeile der Cursorposition an.

`Datentyp(x)`
gibt den Datentyp von `x` an. Mögliche Antworten sind `Objekt`, `Klasse`, `Textstring`, `Routine` und `nichts`.

`Eingabekanal(k)`
aktiviert den Kanal `k` für die Eingabe und deaktiviert alle anderen Eingabekanäle. Mögliche werte von `k` sind null für die Tastatur oder eins für eine externe Datei.

`erzeuge(k)`
erzeugt eine Instanz der Klasse `k`. Wenn ein Objekt erzeugt wurde, wird die Eigenschaft `erzeugen` des Objekts aufgerufen und die Fuinktion gibt das Objekt zurück. Wenn kein Objekt erzeugt werden konnte, ist die Antwort null.

`Kinder(o)`
gibt die Anzahl der Kinder des Objekts `o` an.

`kopiereFeld(eins, zwei, n)`
kopiert `n` Bytes von Feld `eins` nach Feld `zwei`.

`lade()`
lädt einen zuvor auf eine Datei gespeicherten Spielstand und setzt das Programm im geladenen Zustand fort. Gibt null zurück, wenn das Laden fehlschlug.

`ladeFeld(f, n, Dateiname)`
lädt `n` Bytes von einer externen Datei `Dateiname` und schreibt sie auf das Feld `f`. Gibt die Anzahl der erfolgreich gelesenen Bytes zurück.

`ladeZug()`
lädt einen zuvor im Speicher der *Z*-Maschine abgelegten Spielstand. Dies wird für das Rückgängigmachen von Spielzügen verwendet.

`lese(f, max)`
liest maximal `max` Zeichen auf das Bytefeld `f`.

`leseArg(n)`
liest den Wert des `n`-ten an die Routine übergebenen Arguments.

`leseEintrag(b, e)`
ermittelt den Wert des Eintrags `e` im Befehl `b`.

`löscheFeld(f, n)`
setzt `n` Bytes des Feldes `f` auf null.

`löscheFenster(f)`
löscht das Fenster `f` und setzt den Cursor in dessen obere linke Ecke. Das Fenster `f` kann 0 für das Textfenster, 1 für das Statusfenster oder –1 für beide Fenster sein. (Im letzten Fall verschwindet das obere Fenster.)

`Mutter(o)`
gibt das Mutterobjekt von `o` an, das auch `nichts` sein kann.

`num(x)`
gibt den Wert von `x` als Dezimalzahl aus.

`schreibeArg(n, Wert)`
belegt das `n`-te an die Routine übergebene Argument mit dem `Wert`.

`Schriftstil(s)`
setzt den Schriftstil für den nachfolgenden Text auf `s`. Der Stil kann `fett`, `kursiv`, `monospace`, `invers` oder null für normale Schrift sein.

`Schwester(o)`
gibt die Schwester des Objekts `o` an, die auch `nichts` sein kann.

`speichere()`
speichert den momentanen Spielstand auf eine externe Datei. Die Antwort ist null, wenn das Speichern fehlschlug.

`speichereFeld(f, n, Dateiname)`
speichert `n` Bytes des Feldes `f` auf eine externe Datei `Dateiname`.

`speichereZug()`
speichert den momentanen Spielstand im Speicher der *Z*-Maschine, um ihn später mit `ladeZug()` wiederherstellen zu können.

`splitte(Textfeld, Wortfeld, max)`
splittet das mit `lese` eingelesene `Textfeld` in maximal `max` Wörter auf, die im `Wortfeld` abgelegt werden. Gibt die Anzahl der gefundenen Wörter zurück.

`Statusfenster(Z)`
erzeugt ein Statusfenster am oberen Rand des Bildschirms, das `Z` Zeilen hoch ist. das Fenster ist zunächst unsichtbar und muss zum Beschreiben erst mit `aktivesFenster(1)`aktiviert werden. Text im Statusfenster wird nicht automatisch invers ausgegeben.

`Text(x)`
gibt den Text `x` aus.

`Taste()`
wartet, bis eine Taste gedrückt wird und gibt dann den ZSCII-Wert dieser Taste an.

`Textfarbe(vg, hg)`
setzt die Hintergrundfarbe des nachfolgenden Textes auf `hg` und die Vordergrund-, das heißt, die Textfarbe auf `vg`.

`TextInFeld(t, f)`
schreibt den Text `t` auf das Byte-Feld `f` und gibt die Länge des Texts zurück.

`Textumleitung(f)`
leitet die Textasusgabe vorübergehend auf das Feld `f` um. Wenn `f` null ist, wird die Umleitung ausgeschaltet.

`Tochter`
gibt die Tochter des Objekts `o` an, das auch `nichts` sein kann.

`verbleibend(k)`
ermittelt, wieviele Instanzen der Klasse `k` noch erzeugt werden können.

`vernichte(o)`
vernichtet das Objekt `o`, wenn es noch im Spiel ist. dabei wird die Eigenschaft `vernichten` aufgerufen.

`Vokabel(x)`
schreibt die Vokabel an der Adresse `x`.

`Wortumbruch(w)`
schaltet den Wortumbruch im oberen Fenster ein (`w==1`) oder aus (`w==0`).

`Zufall(x, ...)`
Wenn mehrere Argumente übergeben werden, ermittelt die Funktion zufällig eines der Argumente als Antwort. Ansonsten bestimmt die Funktion eine Zufallszahl im Bereich von eins bis `x`, wenn `x` positiv ist, setzt die Saat des Zufallszahlengenerators auf `-x`, wenn `x` negativ ist, oder initialisiert den Zufallszahlengenerator des Interpreters, wenn `x` null ist.

Anhang B:

Überblick über die Bibliothek

(Nicht aktualisierte Arbeitsfassung!)

Das Modell der Spielwelt

Räume und Richtungen

Die Spielwelt besteht aus Räumen, Objekten, die keine Mutter haben und die zur Klasse `Raum` gehören.

Diese Räume sind miteinander durch Richtungen verbunden. Richtungen sind Objekte der Klasse `Himmelsrichtung`, die sich ein einem bestimmten Objekt, der `Windrose` befinden.

Die Ausgänge eines Raumes werden in dessen Eigenschaft `Ausgänge` beschrieben. Das Richtungsobjekt wird als Argument und als Wert eines impliziten `jenach` übergeben, die Antwort ist entweder ein Raum oder ein Verbindungsobjekt oder einer der Werte falsch oder wahr. Wenn die Antwort falsch ist, so wird eine Standard-Absage geschrieben, entweder `aRaum.Ausänge(nichts)` oder ein Lib-Text.

Der wichtigste Befehl zur Fortbewegung lautet `gehen` und erwartet eine Himmelsrichtung. Ein weiterer Befehl, `betreten` erlaubt das Bewegen zu benachbarten Räumen, wenn diese `bekannt` sind.

Raumbeschreibungen

Je nachdem, ob ein Raum besucht worden ist oder nicht, in welchem Modus (knapp, normal, ausführlich) sich der Spieler befindet und welcher Raumbeschreibungsstil gewünscht ist, wird zunächst der `Name` des Raums, eventuell gefolgt von einem Tochterobjekt, auf dem sich der Spieler befindet („auf dem Thron“), ausgegeben, danach folgt die `Beschreibung` des Raums.

Es folgt eine Liste der Gegenstände im Raum, die weder zur Klasse `Deko` gehören noch das Attribut `verborgen` haben. Zunächst werden alle Objekte, die die Eigenschaft `Immer`

haben, gelistet. Der Text aus dieser Eigenschaft wird ausgegeben. Danach werden alle nicht bereits erwähnten Objekte mit der Eigenschaft `Erst` ausgegeben. Jedes Objekt bekommt einen eigenen Absatz, eventuelle Inhaltsangaben werden angehängt.

Danach wird die Eigenschaft `Inhalt` des Raums aufgerufen, die alle nicht bereits erwähnten Objekte in eine Liste packt. Die globale Variable `außerdem` ist wahr oder falsch, jenachdem, ob vorher bereits etwas ausgegeben wurde.

Wenn es keine weiteren Objekte gibt, wird die Eigenschaft `kein_Inhalt` aufgerufen. Wenn bislang nichts ausgegeben wurde, und es weitere Objekte gibt, wird die Standardmeldung „Du siehst hier.../Außerdem siehst du hier...“ ausgegeben.

Inventar, Beschreibungen von Gegenständen

Gegenstände werden mit ihrer Eigenschaft `Name` benannt, die zusammen mit der Eigenschaft `Genus` an den jeweiligen Fall angepasst ausgegeben wird. Der Genus enthält nicht nur das grammatische Geschlecht, sondern auch Informationen, etwa ob es sich um einen Eigennamen handelt oder ob immer der bestimmte Artikel benutzt werden soll.

Beim Untersuchen eines Gegenstands wird die Eigenschaft `Beschreibung` ausgegeben, eventuell gefolgt von einer Inhaltsangabe. Wird nichts ausgegeben, so erscheint bei `schließbaren` Objekten „Es ist auf/zu“, bei `schaltbaren` „Es ist an/aus“ oder einfach „Du siehst nichts Besonderes“.

In Listen wie dem Inventar oder in Raumbeschreinungen kann man Objekte mit Zusätzen wie „(halbvoll)“ versehen oder den Namen des Objekts ändern. Dazu kann man die Eigenschaften `Vor_Liste` und `Nach_Liste` benutzen. Ein wahrer Rückgabewert bricht die Ausgabe des Namens hier ab. Zum Schluss kann man allgemein gültige Angaben im Einhänger `Zusatzinfo` machen.

Das Inventar wird in zwei Formaten angeboten: hoch (als Liste) und quer (als Satz).

Aufheben von Gegenständen

Alle Objekte, die nicht eines der Attribute `fest` oder `Person` besitzen oder zu einer der Klassen `Raum` oder `Deko` gehören, lassen sich vom Spieler aufheben und mit sich herumtragen. Die Befehle hierzu heißen `nehmen` und `ablegen`.

Für eine realistischere Spielwelt stehen die Eigenschaften `Gewicht` und `Volumen` zur Verfügung, die der Spieler verwenden kann oder nicht. Die Maximalwerte sind diejenigen des momentanen Spielerobjekts. Außerdem hat jeder Behälter und jede Ablage eine `Kapazität`, genau wie der Spieler.

Dekorationen

Räume können, wie alle Objekte, Vokabular besitzen. Dieses bezieht sich auf den Raum selbst, nicht wie in Inform auf unwichtige Objekte. So kann man sich z.B. mit `betreten` auf Räume beziehen.

Unwichtige Details in der Raumbeschreibung können mit der Klasse `Unwichtig` erzeugt werden, die nichts anderes tut, als Vokabular abzufangen.

Behälter, Ablagen und Unterobjekte

Objekte, die sichtbare Tochterobjekte haben, müssen entweder ein `Behälter` oder eine `Ablage` sein, jedoch nie beides gleichzeitig.

Ein Behälter ist ein Objekt, in dem sich andere Objekte befinden. Damit man die Objekte sehen kann, darf der Behälter nicht `geschlossen` sein oder er muss das Attribut `transparent` besitzen. Im letzten Fall können die Objekte im Behälter jedoch nicht berührt werden. Die Befehle lauten `hineinlegen` und `herausnehmen`.

Eine Ablage ist ein Objekt, auf dem sich andere Objekte befinden, die immer dann sichtbar und berührbar sind, wenn es die Ablage selbst ist. Die Befehle lauten hier `daraufflegen` und `herunternehmen`.

Eine weitere Möglichkeit, dass Tochterobjekte sichtbar sind, ist, dass eine `Person` diese Objekte bei sich trägt. Wenn diese Person `transparent` ist, kann man diese Objekte sehen. Normalerweise kann man Objekte, die eine Person bei sich trägt, nicht anfassen oder manipulieren.

Die Eigenschaft `Inhalt` kann eine Routine beinhalten, die eine Liste des Inhalts des Objekts ausdruckt, wie sie in der Raumbeschreibung oder beim Untersuchen des Objekts auftaucht. Hat ein Behälter, eine Ablage oder eine Person diese Eigenschaft nicht, wird die Standardmeldung, etwas wie „In/auf dem Schrank siehst du...“, ausgegeben.

Die Eigenschaft `kein_Inhalt` erlaubt es, einen Text auszugeben, wenn das Objekt keine Kinder hat, also leer oder frei ist. Fehlt diese Eigenschaft, wird nichts ausgegeben.

Zusätzlich können Objekte Unterobjekte besitzen, die in der Eigenschaft `Unterobjekte` als Liste angegeben werden. Diese sind sichtbar und berührbar wie die Hauptobjekte, jedoch nicht ihre Kinder im Objektbaum. Diese Objekte werden auch nicht bei der Inhaltsangabe der Objekte berücksichtigt.

Türen

Objekte mit dem Attribut `Tür` verbinden zwei Räume. Eine Tür im Sinne von Zeta muss nicht unbedingt eine Tür sein, es kann sich auch um eine Brücke oder einen Pfad handeln. Damit man durch eine Tür gehen kann, darf sie nicht `geschlossen` sein.

Eine Tür hat einen `Zielraum`, eine Eigenschaft, die beschreibt, wohin diese Tür führt. Um das Gehen in eine Richtung abzufangen, kann eine Tür auch die Eigenschaft `Richtung` besitzen. Der Ausgang des Raums muss dann auf die Tür verweisen.

Die Befehle zum Öffen und Schließen von Türen und Behältern oder anderen Objekten, die das Attribut `schließbar` besitzen, heißen `öffnen` und `schließen`. Außderdem können Türen `betreten` werden.

Schlüssel und Schlösser

Objekte, die das Attribut `abschließbar` haben, können abgeschlossen werden. Dann haben sie die Attribute `geschlossen` und `abgeschlossen`.

Objekte, die die Eigenschaft `Schlüssel_für` definieren, sind Schlüssel. Mit ihnen können abschließbare Objekte aufgeschlossen werden. Die Eigenschaft `Schlüssel_für` kann eine Liste von Objekten enthalten. Die Befehle hier sind `aufschließen` und `abschließen`.

Betretbare Objekte und Fahrzeuge

Manche Objekte haben das Attribut `betretbar`, dann kann der Spieler auf ihnen sitzen, stehen oder liegen, wenn sie eine Ablage sind oder in ihnen, wenn sie Behälter sind. Die Variable `Sitzposition` ist einer der Befehle die Werte `sitzen`, `stehen` oder `liegen` oder null wenn seine Position (wie in einem Raum) egal ist. Die Befehle lauten hier `betreten` und `verlassen`. Die Spezialfälle `stehen`, `sitzen` und `liegen` werden in `betreten` mit der jeweiligen Sitzposition umgewandelt.

Betretbare Objekte werden in der Raumbeschreibung mit einem eigenen Absatz berücksichtigt, der mit der Eigenschaft `Innenansicht` definiert wird.

Ein Objekt, das das Attribut `Fahrzeug` hat, bewegt sich mit dem Spieler, wenn er darauf sitzt. Der passende Befehl heißt `fahren`, er lässt im Allgemeinen jede Bewegung zu. Änderungen müssen vom Autor abgefangen werden. Wenn der Spieler in einem Fahrzeug sitzt und eine Richtung eingibt, wird der Befehl in „fahre mit dem Fahrzeug in diese Richtung“ umgewandelt. Ein Reittier würde auch als Fahrzeug definiert.

Fahrzeuge können von einem Raum in einen anderen verschoben werden. Dann wird der Befehl `schieben` ausgeführt, der wie `fahren` generell alles das erlaubt, was der Spieler auch zu Fuß darf. Ausnahmen müssen vom Autor berücksichtigt werden. Ein Fahrzeug muss nicht unbedingt betretbar sein.

Schaltbare Objekte

Objekte, die das Attribut `schaltbar` besitzen, können ein- und ausgeschaltet werden. Wenn sie eingeschaltet sind, besitzen sie das Attribut `eingeschaltet`. Die Befehle dazu lauten `einschalten` und `ausschalten`. Was mit den eingeschalteten Objekten passiert, muss der Autor selbst definieren.

Kleidung

Objekte, die der Spieler anziehen kann, werden mit dem Attribut `Kleidung` versehen. Wenn der Spieler ein Kleidungsstück angezogen hat, bekommt es das Attribut `angezogen`. Die passenden Befehle lauten `anziehen` und `ausziehen`.

Gesprächsthemen und Stichpunkte

Mit den Befehlen `fragen` und `erzählen` kann man Personen über bestimmte Themen befragen, mit `nachschlagen` kann man Stichpunkte in Notizbüchern oder Lexika nachsehen.

Das Thema ist dabei ein beliebiges Objekt, das nicht unbedingt in Sicht sein muss. Es kann jedes Thema sein, obwohl Objekte, die das Attribut `Stichpunkt` besitzen, bevorzugt werden. Objekte mit dem Attribut `bekannt` werden unter den Stichpunkten bevorzugt.

Anweisungen an andere Personen

Anweisungen an andere Personen werden in der Eigenschaft `Anweisungen` berücksichtigt. Dabei wird der Pseudobefehl `nicht_verstanden` verwendet, wenn die Eingabe fehlerhaft war.

Mit der Eigenschaft `neues_Verb` können alternative Verben, wahrscheinlich „unmögliche Verben“, die einen Punkt oder ein Komma enthalten, benutzt werden.

Lichtverhältnisse

Helle Räume haben das Attribut `hell`, Objekte, die leuchten sind `Lichtquellen`, ein Synonym für dasselbe Attribut. Der Spieler kann sehen, wenn es im Raum Licht gibt, d.h. wenn der Raum hell ist oder es eine sichtbare Lichtquelle gibt.

Die genauen Regeln sind wie in Inform: Licht breitet sich durch transparente Objekte, nicht geschlosene Behälter und Ablagen aus.

Ein Objekt bekommt Licht wenn es selbst hell ist, eines seiner Kinder Licht verbreitet oder es lichtdurchlässig ist seine Mutter Licht bekommt.

Ein Objekt verbreitet Licht, wenn es selbst hell ist, es lichtdurchläsig ist und eines seiner Kinder Licht verbreitet oder wenn eines seiner Unterobjekte Licht verbreitet. Der Spieler kann sehen, wenn sein Mutterobjekt Licht bekommt.

Dämonen und Zünder

Jedes Objekt kann die Eigenschaft `Dämon` haben, die eine Routine enthält, die am Ende jeden Zugs aufgerufen wird, wenn der Dämon aktiv ist. Ob der Dämon aktiv ist, wird im Attribut `Dämon_Aktiv` festgehalten. Um einen Dämon zu starten oder anzuhalten, kann man die Library-Routinen `Dämon_An(Obj)` und `Dämon_Aus(Obj)` benutzen.

Außerdem kann ein Objekt einen Zeitzünder besitzen, der mit `Setze_Zünder(Obj, Zeit)` aktiviert wird und bei Ablaufen der Zeit gezündet wird. Die verbleibende Zeit wird in der Eigenschaft `Restzeit` gespeichert, die Eigenschaft, die beim Zünden aufgerufen wird, ist `Zündung`. Der Zünder kann vor Erreichen der Zündung mit `Lösche_Zünder(Obj)` deaktiviert werden.

Eine weitere Eigenschaft, `Jede_Runde`, wird immer dann aufgerufen, wenn das betreffende Objekt in Sicht ist und kann für Textausgaben genutzt werden.

Der Spieler

Der Spieler wird durch das Objekt, das in der Variable `Spieler` festgehalten wird, repräsentiert, normalerweise das Objekt `Protagonist`.

Initialisierung

Jedes Objekt kann die Eigenschaft `Init` definieren, die zu Beginn für jedes Objekt einmal aufgerufen wird. Dies geschieht, bevor die Routine `Anfang` aufgerufen wird.

Ändern der Standardmeldungen

Die Standardmeldungen der Bibliothek können im Bibiotheksobjekt `Texte` überschrieben werden. Für Fehlermeldungen des Parsers gibt es die Rubrik `nicht_verstanden`.

Ablauf eines Zuges

Nach der erfolgreichen Analyse der Eingabe durch den Parser werden folgende Stadien durchlaufen:

Nachparser.
- Die globale Routine `Nachparser` wird aufgerufen.

Vorher.
- `Vorreaktion` der Lib-Objekte
- `Vorreaktion` des momentanen Aufenthaltsraums
- `Vorreaktion` aller Objekte in Sichtweite
- `Vorher` des ersten Objekts `aObj`
- `Vorher` von `aObj2`, geprüft auf den Pseudobefehl

Während.
- `Aktion` des jeweiligen Befehls
- `Meldung` des ersten Objekts `aObj`
- `Meldung` von `aObj2`
- `Erfolg` des Befehls wird bekannt gegeben

Nachher.
- `Nachher` des ersten Objekts `aObj`
- `Nachher` von `aObj2`
- `Nachreaktion` aller Objekte in Sichtweite.
- `Nachreaktion` des momentanen Aufenthaltsraums.
- `Nachher` der Lib-Objekte

Die Rückgabe von `wahr` oder einem anderen von null verschiedenen Wert aus einer dieser Routinen bricht die Kette der Aktionen sofort ab. Es wird davon ausgegangen, dass der Grund des negativen Ergebnisses dem Spieler mitgeteilt wurde.

In jedem Fall werden nach Durchführung der Aktion die folgenden Dinge durchgeführt:

- Die globale Variable `Züge` wird aktualisiert.
- `Stunden` und `Minuten` werden aktualisiert.
- Alle Dämonen und Zeitzünder werden gehandhabt.
- Alle `Jede_Runde`-Eigenschaften werden ausgeführt.
- Objekte bekommen das Attribut `bewegt`.
- Die Punkte werden überprüft und Änderungen ausgegeben.

Meta-Befehle werden direkt ausgeführt, ohne Möglichkeit sie abzufangen, da sie als nicht zur Spielwelt gehörend betrachtet werden.

Ein anderer Befehl kann ausgeführt werden, wenn die Routine `Ausführung(xBefehl, xObj, xObj2, stumm)` aufgerufen wird. Jedes der Objekte kann `nichts` sein. Wenn `stumm` einen wahren wert hat, wird der Erfolg des Befehls nicht bekannt gegeben. Fehlschläge des Befehls werden immer bekannt gegeben. Die Antwort von `Ausf` ist `falsch`,

wenn der Befehl nicht abgebrochen wurde, `wahr` sonst.

Der Parser

Überprüfen, ob ein Wort der Eingabe entspricht

Bevor der Text analysiert wird, steht die Variable zur Markierung des nächsten Worts, der Wortzähler `wn`, auf dem ersten zu analysierenden Wort.

Danach wird für alle Objekte die Eigenschaft `Parser` aufgerufen, die die nachfolgenden Wörter analysieren kann, und einen der folgenden Werte zurückgibt: `falsch`, um das Wort nicht zu berücksichtigen, oder einen anderen Wert, um das Wort in die Liste aufzunehmen. `1` fährt mit der Analyse fort, um die normalen Regeln anzuwenden, `2` bricht die Analyse ab und `3` nimmt das Wort auf und markiert es als einen Plural (entspricht `pp`). Der Wortzähler muss auf dem ersten Wort, das nicht zum gefundenen Objekt gehört, stehen gelassen werden, wenn ein Treffer vorliegt.

Danach werden die Eigenschaften `Parser` der Bibliotheksobjekte für jedes Objekt ausgeführt, mit den selben Rückgabemöglichkeiten. Schließlich wird jedes Objekt anhand seiner Eigenschaften `Adj` und `Subst` analysiert.

Überprüfen, ob ein Objekt in Frage kommt

Wenn der Befehl `aBefehl` eindeutig bestimmt ist, und ein Objekt gefunden wurde, wird überprüft, ob dieses Objekt benutzt werden kann. In der Regel wird geprüft, ob es sichtbar oder berührbar ist.

Dazu wird der Eintrag `ungültig` des Befehls betrachtet, eine Routine, die entweder `falsch` zurückgibt, um das Objekt zuzulassen, oder `wahr`, eine Fehlernummer oder eine Fehlermeldung, um es auszuschließen.

Wenn es nur ein Objekt in der Liste der möglichen Objekte gibt und ein Fehler auftritt, so wird entweder die Fehlermeldung ausgegeben oder die Meldung mit der passenden Fehlernummer.

Die Variable `Plural_Modus` ist gesetzt, wenn überprüft werden soll, ob ein Objekt in einer Liste von Objekten mit „alles“ eingeschlossen werden soll. So ist „nimm Nussbaum“ legal. Dass der Baum nicht genommen wird, wird in der `Aktion` des Befehls abgefangen. Trotzdem soll bei „nimm alles“ der Baum nicht berücksichtigt werden.

Zwei Objekteigenschaften können die Überprüfung im Vorfeld beeinflussen: Mit der Eigenschaft `verbieten` kann man, wie in `ungültig`, einen wahren Wert zurückgeben, um

das Objekt auszuschließen. Die Eigenschaft `erlauben` kann einen wahren Wert zurückgeben, um das Objekt einzuschließen.

Routinen zur Überprüfung der Gültigkeit sollten keinen Text ausgeben!

Disambiguisierung

Wenn der Parser zwischen mehreren Objekten entscheiden muss, werden alle nicht zugelassenen Objekte aus der Liste gestrichen.

Wenn genau ein Objekt übrig bleibt, so wird dieses Objekt genommen.

Wenn mehrere Objekte übrig bleiben, so wird der Eintrag `Priorität` aufgerufen, eine Routine, die die Objekte auf einer Skala von null (schlecht) bis zehn (gut) bewertet. So können Objekte mit dem Attribut `Kleidung` beim Anziehen bevorzugt werden und so weiter.

Ist immer noch keine eindeutige Wahl getroffen, so kann die Eigenschaft `Disambig` der Objekte herangezogen werden, die ebenfalls eine Wertung zurückgibt.

Wenn ein Objekt gefunden wurde, wird eine Annahme „(mit dem Brotmesser)“ gedruckt und dieses Objekt benutzt. Wenn hier keine eindeutige Entscheidung getroffen werden konnte, fragt der Parser nach. Die Ausgabe der Annahme kann durch Setzen der globalen Vaiable `stumm` unterdrückt werden.

Wenn kein Objekt übrig bleibt, so wird das Objekt mit der höchsten Fehlernummer genommen, Fehlermeldungen zählen 1000 in dieser Betrachtung. Wenn mehrere Objekte übrigbleiben, wird disambiguisiert wie oben beschrieben, wenn kein Objekt hervorsticht, fragt der Parser nach, wenn die Fehlernummer größer als eins ist. Ansonsten wird ausgegeben „Du siehst hier kein...“, wobei der Wortlaut des Spielers verwendet werden sollte, um nichts preiszugeben.

Hat ein Objekt eindeutig die höchste Fehlernummer, so wird zunächst versucht, den Fehler zu beheben. Dies geschieht, wenn die Fehlernummer größer als eins ist und der Befehl den Eintrag `implizit` hat, eine Routine, die versucht, das Objekt in die Reichweite des Spielers zu bringen, z.B. ein Objekt aufzuheben. Danach wird die Bedingung noch einmal geprüft, und wenn sie jetzt erfüllt ist, wird das Objekt genommen.

Ansonsten wird – wie oben beschrieben – eine Fehlermeldung für dieses Objekt ausgegeben.

Die Variable `stumm` kann gesetzt werden, um die Meldung beim Disambiguisieren „(den roten Fisch)“ oder die Erfolgsmeldung bei einigen implizit aufgerufenen Befehlen zu unterdrücken.

Unterscheiden der Objekte

Alle Befehlseinträge, die `aObj2` statt `aObj` behandeln, haben eine `2` angehängt: `ungültig2`, `Priorität2` usw. Dasselbe gilt für Pseudobefehle: `herausnehmen2`, `fragen2`.

Lösungen der Aufgaben

• **1** Man definiert einfach zusätzliche Objekte der Klasse `Himmelsrichtung` als Töchter des Objekts `Windrose`. Um die Richtungen an Land und an Bord zu unterscheiden, markieren wir die neuen Richtungen mit dem Attribut `an_Bord`.

```
Attribut an_Bord;

Himmelsrichtung Steuerbord in Windrose {
    Name = "Steuerbord";
    Genus = n;
    Subst = 'steuerbord' n, 'sb' n, 'stb';
    ist an_Bord;
}
```

Um die Fehlermeldung etwas anzupassen, können wir den Standardtext „Du kannst nicht in diese Richtung gehen“ ändern:

```
LibObjekt in ZetaLib {
    Texte(n) = {
        gehen: wenn (n == 1) {
        wenn (aObj == raus || aObj == rein
            || aObj == hoch || aObj == runter) antworte;
        wenn (aRaum ist an_Bord && !(aObj ist an_Bord))
            stop "Hier an Bord verwendet man die
                Richtungen Steuerbord, Backbord, voraus
                und achtern, du Landratte.";
        wenn (!(aRaum ist an_Bord) && aObj ist an_Bord)
            stop "Hast du wieder am Rum genascht? Zu Land
                gibt's ", kein_Objekt(aObj, Akkusativ),
                ", benutze die Himmelsrichtungen.";
        }
}
```

Räume an Bord eines Schiffes müssen ebenfalls das Attribut `an_Bord` besitzen.

• **2** Zunächst müssen wir die neuen Eigenschaften definieren, für jede Richtung eine. Die Eigenschaft `kein_Weg` wird aufgerufen, wenn es keinen Ausgang in diese Richtung gibt.

```
Eigenschaft nach_N;
Eigenschaft nach_NO;
...
Eigenschaft + kein_Weg;
```

Nun definieren wir eine Klasse `Ort`, der alle Räume angehören. In der Eigenschaft `Ausgänge` dieser Klasse leiten wir nun die Anfrage an die einzelnen Richtungseigenschaften um. Zur Zuordnung der Objekte und Eigenschaften benutzen wir hier eine Routine. (In Inform wird die assiziierte Eigenschaft beim Richtungsobjekte gespeichert, aber dann müssten wir hier alle Richtungsobjekte ersetzen.)

```
Routine Richtungseigenschaft(o) {
    jenach (o) {
        Norden: antworte nach_N;
        Nordosten: antworte nach_NO;
        ...
        sonst: antworte kein_Weg;
    }
}

Klasse Ort {
    beerbt Raum;
    Ausgänge(r) = {
        lokal Ziel, e = Richtungseigenschaft(r);
        antworte aRaum.e();
    }
}
```

Die Rückgabewerte sind wie bei `Ausgänge`, un eine typische Raumdefinition könnte folgende Eigenschaften haben:

```
nach_N = Hängebrücke;
nach_NO = "Dort geht es in die Schlucht hinunter.";
nach_NW = { antworte selbst.nach_NO(); }
nach_S = {
    wenn (Amulett in Spieler && Amulett hat angezogen)
        antworte Geheimgang;
    stop "Eine magische Kraft hält dich zurück.";
}
```

• 3

Hier fangen wir am besten den Standardtext ab, es ist `LibText(gehen, 1)`. In einer Schleife über alle Objekte in der `Windrose` wird geprüft, ob der Ausgang in diese Richtung ein Objekt ist.

```
Variable sondieren = falsch;

LibObjekt in ZetaLib {
    Texte(n) = {
        lokal o;
```

```
            gehen:
            wenn (n != 1) antworte;
            "Du kannst nicht nach ",
                Objekt_ohne_Art(aObj), " gehen";
            sondieren = wahr;
            o = Tochter(Windrose);
            lösche_Liste();
            solange (o) {
                schreibeArg(255, o);
                wenn (Datentyp(aRaum.Ausgänge(o))==Objekt)
                    setze_auf_Liste(o);
                o = Schwester(o);
            }
            sondieren = falsch;
            wenn (Liste.Anzahl == 0) stop ". (Du kannst
                im Moment nirgendwohin gehen, so wie's
                aussieht.)";
            stop ", nur nach ",
                Objekt_ohne_Art(Liste), ".";
        }
    }

    Routine stummer_Text(x) {
        wenn (sondieren) antworte;
        x();
    }
```

Das Problem hierbei ist, dass man beim Senden der Mitteilung `Ausgänge` nicht weiß, ob in der Routine Text ausgegeben wird. Vor der Schleife wird daher die Variable `sondieren` gesetzt, um anzuzeigen, dass `Ausgang` nicht benutzt wird, um tatsächlich zu gehen, sondern nur um zu sehen, ob es überhaupt einen Ausgang in diese Richtung gibt. Alle Textausgeban in `Ausgänge` (außer die generellen Absagen unter `nichts`) müssen daher diese Variable prüfen. Wenn die Antwort nur ein statischer Text ist, kann man die Routine `stummer_Text` verwenden:

```
    Ausgänge = {
        Norden: antworte Gang;
        Süden: stop stummer_Text("Der Weg in diese Richtung
            ist blockiert, dort ist der Stollen eingestürzt.");
    }
```

- **4**

- **5** Dazu verwendet man die Eigenschaft `gibts_in` wie bei den Schiebekulissen und macht aus `Zielraum` eine Routine, die den passenden Raum ausgibt:

```
Objekt -> Wohnzimmertür {
    Name = "Wohnzimmertür";
    Subst = 'tür' f, 'wohnzimmertür' f;
    gibts_in = Wohnzimmer, Diele;
    Zielraum = {
        wenn (aRaum==Diele) antworte Wohnzimmer;
        antworte Diele;
    }
    ist Verbindung, schließbar, geschlossen;
}
```

Für ein Spiel mit vielen Türen bietet es sich an, eine Klasse Tür zu schreiben:

```
Klasse Tür {
    Subst = 'tür' f;
    Zielraum = {
        wenn (aRaum == selbst.gibts_in[0])
            antworte selbst.gibts_in[1];
        antworte selbst.Räume[0];
    }
    ist Verbindung, schließbar, geschlossen;
}
```

Nun kann man bequem Türen definieren. Die beiden Räume, die sie verbindet, stehen in der Eigenschaft `gibts_in`, die immer zwei Räume enthalten sollte und die bewirkt, dass die Tür von beiden Seiten sichtbar ist. Dieselbe Wohnzimmertür also nochmal:

```
Tür -> Wohnzimmertür {
    Name = "Wohnzimmertür";
    Subst = 'wohnzimmertür' f;
    gibts_in = Wohnzimmer, Diele;
}
```

• **6** Man muss nur zu jedem benachbarten Raum eine `Verbindung` definieren, die die Vokabeln des Nachbarraums versteht. So kann man sich mit „gehe ins Wohnzimmer“ oder „betrete die Kirche“ zwischen den Räumen fortbewegen.

Da das Definieren vieler Verbindungen mühselig ist, hier ein Beispiel für eine Objektklasse für standardisierte Verbindungen:

```
Klasse Raumverbindung {
    gibts_in = {
        wenn (selbst hat Nachbarräume) {
            lokal n = selbst.#Nachbarräume;
            solange (n--)
                wenn (selbst.Nachbarraum[n] == aRaum)
                    antworte wahr;
```

```
            }
        }
        Vorher = {
            betreten, gehen:
            sonst: stop "Das ";
        }
        ist Verbindung, verborgen, fest;
    }
```

Zu jedem Raum gibt es dann nur ein Verbindungsobjekt, das mit `gibts_in` in die Nachbarräume verschoben wird, die durch eine Objektliste in dessen Eigenschaft `Nachbarräume` abgelegt werden. Zum Beispiel:

```
    Raumverbindung {
        Name = "Flur";
        Genus = m;
        Subst = 'flur' m, 'gang' m, 'korridor' m;
        Nachbarräume = Wohnzimmer, Esszimmer, Bad,
            Gästetoilette, Küche, Kinderzimmer,
            Elternschlafzimmer, Gästezimmer, Vorraum;
    }
```

Die Himmelsrichtungen werden damit bedeutungslos, wer sie gar nicht erst als Vokabel zulassen möchte, kann die Richtungsobjekte ersetzen oder deren Definition in der Bibliothek mit `keine_Himmelsrichtungen` unterbinden.

• **7** Für die Lösung definieren wir eine Klasse für Räume bei der Schlucht, dort kann dan die Planke abgelegt werden. Die Eigenschaft `andere_Seite` enthält den Raum auf der anderen Seite der Schucht, per Default eine Absage.

```
    Klasse Schluchtraum {
        beerbt Raum;
        andere_Seite = "Du kannst hier nicht über die
            Schlucht gehen.";
    }

    Objekt Planke {
        Name = "Holzplanke";
        Genus = f;
        Adj 'lang', 'stabil';
        Subst = 'planke' f, 'holzplanke' f, 'brett' n,
            'holzbrett' n;
        Beschreibung = {
            wenn (selbst ist Verbindung) {
                stop "Die Planke liegt quer über der
                    Schlucht.";
            } sonst "Lang und stabil.";
        }
```

```
        Zielraum = {
            antworte aRaum.andere_Seite();
        }
        Vorher = {
            nehmen: mache selbst nicht Verbindung;
            sonst: wenn (selbst in aRaum
                && aRaum beerbt Schluchtraum)
                mache selbst Verbindung;
        }
        Meldung = {
            hinlegen:
                wenn (!(aRaum beerbt Schluchtraum))
                    antworte falsch;
                mache selbst Verbindung;
                stop "Du legst die Planke über die
                    Schlucht, so dass sie jetzt beide
                    Seiten miteinander verbindet.";
        }
    }
```

Die Planke muss vor dem Betreten noch einmal explizit das Attribut `Verbindung` bekommen, weil ein gescheitertes Nehmen – vielleicht, weil der Spieler bereits die Hände voll hatte – die Planke ohne dieses Attribut herumliegen lassen könnte. (Das hätte man auch mit einem Dämonen oder Ähnlichem machen können, aber diese werden erst später behandelt.)

• **8** Man könnte in `Vorher` das Auf- und Abschließen beim Schlüssel abfangen und die Eigenschaft `Schlüssel_für` mit dem jeweiligen Objekt überschreiben:

```
    Objekt General {
        Name = "Generalschlüssel";
        Genus = m;
        Adj = 'klein', 'silbern';
        Subst = 'schlüssel' m, 'generalschlüssel' m,
            'general' m;
        Beschreibung = "Dieser kleine silberne Schlüssel
            ist eines der Insignien deiner Macht als
            Hausmeister.";
        Schlüssel_für = nichts;
        Vorher = {
            hinlegen: stop "Niemals!";
            aufschließen2, abschließen2:
                selbst.Schlüssel_für = aObj;
        }
    }
```

Wichtig ist hier, nicht `stop` zu verwenden, sondern die Routine mit einer falschen Antwort zu verlassen, um das Aufschließen zuzulassen. Wesentlich einfacher ist es aber, eine Routine für `Schlüssel_für` zu verwenden:

```
Schlüssel_für = { antworte wahr; }
```

• **9** Hier definieren wir für jeden Raum, von dem aus man sich mit dem Boot fortbewegen kann, eine Eigenschaft `Kanu_Ausgänge`, die analog zu `Ausgänge` funktioniert. Diese Eigenschaft wird im Code des Kanus überprüft. Zusätzlich bekommen Räume, die vollständig im Wasser liegen, das Attribut `Wasser`.

```
Attribut Wasser;
Eigenschaft Kanu_Ausgänge;

Objekt Kanu {
    Name = "Kanu";
    Genus = n;
    Subst = 'kanu' n, 'boot' n, 'kanadier' m;
    Beschreibung = "Ein kleines, leichtes Kanu.";
    Immer = "Dein Kanu steht hier.";
    Vorher = {
        lokal n;
        verlassen:
            wenn (aRaum ist Wasser) stop "Nicht hier.";
        fahren:
            wenn (Spieler nichtin selbst)
                stop "Dazu musst du ins Kanu steigen.";
            wenn (Paddel nichtin Spieler)
                stop "Du hast nichts zum Paddeln.";
            wenn (aRaum hat Kanu_Ausgänge) {
                schreibeArg(255, aObj2);
                n = aRaum.Kanu_Ausgänge(aObj2);
                wenn (Datentyp(n) == Objekt) {
                    wenn (n istnicht Wasser)
                        "Du erreichst das Ufer und setzt
                        dein Kanu an Land.\p";
                    gehe_zu(n, 0, selbst);
                }
            }
            wenn (!n) stop "In diese Richtung kannst
                du nicht paddeln.";
            stop;
    }
    Sitzpositionen = sitzen, liegen;
    ist Fahrzeug, betretbar, Behälter;
}

Objekt Paddel {
    Name = "Paddel";
    Genus = n;
    Adj = 'hölzern';
    Subst = 'paddel' n;
    Beschreibung = "Ein hölzernes Paddel.";
```

```
}
```

Mit `schreibeArg(255, aObj2)` wird das implizite „jenach“ für `Kanu_Ausgänge` ermöglicht.

• **10** In diesem Fall lassen wir nur Tochterobjekte des Hilfsobjekts `Galaxis` als Einträge zu.

```
Objekt Sternenatlas {
    Name = "Sternenatlas";
    Genus = m;
    Subst = 'atlas' m, 'sternenatlas' m,
        'sternatlas' m, 'sternkarte' f,
        'karte' f, 'sternenkarte' f;
    Beschreibung = "Ein Sternenatlas zum
        Nachschlagen aller bekannten Sterne
        im westlichen Spiralarm der Galaxis.";
    Vorher = {
        nachschlagen:
        wenn (aObj2 nichtin Galaxis)
            stop "Über diesen Stern findest du
            nichts. Vielleicht ist es keiner.";
        """";
        Schriftstil(kursiv);
        aObj2.Name();
        Schriftstil(0);
        ": ", aObj2.Beschreibung(), """";
    }
}

Objekt Galaxis;

Objekt -> {
    Name = "Yewis III";
    Subst = 'yewis' m, 'iii' m;
    Beschreibung = "Die Unterwasserwelt von
        Yewis ...";
    ist Stichwort;
}

Objekt -> {
    Name = "Faghum II";
    Subst = 'faghum' m, 'ii' m;
    Beschreibung = "Faghum II ist die Hauptwelt
        der Allianz von Chalybes ...";
    ist Stichwort;
}
```

• **11** Man könnte die Tür auch mit einem Dämon implementieren, hier verwenden wir einen Zünder.

```
Objekt Schiebetür {
    Name = "Schiebetür";
    Genus = f;
    Adj = 'automatisch';
    Subst = 'tür' f, 'schiebetür' f, 'kartenschlitz' m,
        'schlitz' m, 'codekartenschlitz' m;
    Zündung = {
        wenn (!nicht_sichtbar(selbst)) "\pDie Tür schließt
            sich wieder.";
        mache selbst geschlossen;
    }
    Vorher = {
        öffnen_mit:
            wenn (aObj2==Codekarte)
                stop Ausf(hineinlegen, aObj2, selbst);
        hineinlegen2:
            wenn (selbst istnicht geschlossen)
                stop "Ein kurzer, hoher Piepton ertönt,
                    sonst passiert nichts.";
            mache selbst nicht geschlossen;
            setze_Zünder(selbst, 1);
            stop "Die Tür schiebt sich fast geräuschlos auf,
                als du die Codekarte kurz in den Schlitz
                steckst.";
        öffnen:
            wenn (Codekarte in Spieler) {
                wenn (vorh_Befehl != betreten)
                    "(mit der Codekarte)\n";
                stop Ausf(hineinlegen, Codekarte, selbst);
            }
            stop "Dazu brauchst du deine Codekarte.";
        schließen:
            stop "Das geht automatisch.";
    }
    gibts_in = Reaktorraum, Oberdeck;
    Zielraum = {
        wenn (aRaum == Reaktorraum) antworte Oberdeck;
        antworte Reaktorraum;
    }
    ist Verbindung, geschlossen, schließbar, verborgen;
}
```

Die Abfrage von `vorh_Befehl` bewirkt, dass „(mit der Codekarte)“ nicht gedruckt wird, wenn die geschlossene Tür betreten wird und sie implizit geöffnet wird, da es sonst zwei Klammersätze in einer Reihe gäbe, was zwar nicht falsch, aber auch nicht schön ist.

• **12** Das Problem ist hier wieder, dass man die Eigenschaft `Ausgänge` des momentanen Aufenthaltsraums des Diebs aufrufen müsste, aber nicht weiß, ob hier Textausgegeben wird oder nicht. daher legen wir zu jedem Raum, den der Dieb betreten kann, eine Eigenschaft `NPC_Ausgänge` an, die eine Liste der möglichen Ausgänge für den Dieb enthält. Auf diese Weise kann man die Bewegungsfreiheit des Diebs einschränken.

```
Objekt Dieb {
    Name = "Dieb";
    Genus = m;
    Subst = 'dieb' m;
    Immer = "Ein Dieb lungert hier herum und beäugt dich
        kritisch.";
    Dämon = {
        wenn (selbst.Wartezeit == 0) {
            lokal r = Mutter(Dieb), n;
            wenn (r hat NPC_Ausgänge) {
                n = r.#NPC_Ausgänge;
                n = r.NPC_Ausgänge[Zufall(n) - 1];
            }
            selbst.Wartezeit = Zufall (4) + 1;
            wenn (Datentyp(n) != Objekt) antworte;
            wenn (r == aRaum) "\pDer Dieb schleicht
                weiter und betrachtet dich abschätzig,
                als er den Raum verlässt.";
            verschiebe selbst nach n;
            wenn (n == aRaum) "\pEin verdächtig
                aussehender Herr hat sich gerade in
                den Raum geschlichen. Er inspiziert
                dich und deine Besitztümer.";
        } sonst selbst.Wartezeit--;
    }
    Wartezeit = 5;
    ist Person;
}
```

• **13** Der Hauptdämon ist der einzige, der in der Initialisierung regulär gestartet wird. Er ruft dann in jeder Runde alle anderen aktiven Dämonen auf. Diese Dämonen kann man von Anfang an aktivieren, indem man ihnen das Attribut `aktiv` gibt. Die Routine `Dämon_an` sollte dann nicht mehr aufgerufen werden. Aufpassen muss man, dass sich der Hauptdämon nicht versehentlich selbst aufruft.

```
    Attribut aktiv;

    Objekt {
        Dämon = {
            lokal o;
            alleObjekte o wenn (o ist aktiv
```

```
                && o != selbst && o hat Dämon) o.Dämon();
        }
        init = { Dämon_an(selbst); }
    }
```

Wenn hier neben den neuen Dämonen keine Zünder benutzt werden sollen, könnte man nun die Konstante `max_Zünder` auf eins setzen.

• **14** Es gäbe hier mehrere Lösungen. Eine Möglichkeit ist, die Prioritäten in einer Schleife sortiert auf ein Feld zu schreiben und sie dann in der richtigen Reihgenfolge abzuarbeiten. Bei dieser Methode ist die Anzahl der aktiven Dämonen wieder von einem Feld abhängig.

Hier wird die Objektschleife mehrmals durchlaufen, was die Anzahl der Dämonen nicht einschränkt, aber bei vielen aktiven Dämonen mit unterschiedlichen Prioritäten das Spiel sehr langsam machen kann:

```
Attribut aktiv;

Objekt {
    Dämon = {
        lokal o, jetzt, danach = 32767;
        wiederhole {
            jetzt = danach;
            danach = -32768;
            alleObjekte o wenn (o ist aktiv
                && o != selbst && o hat Dämon) {
                lokal p = o.Dämon_Prio;
                wenn (p == jetzt) o.Dämon();
                wenn (p < jetzt && p > danach) danach = p;
            }
        } bis (danach == -32768);
    }
    init = { Dämon_an(selbst); }
}
```

Die Priorität kann hier jeden beliebigen Wert annehmen, außer den niedrigsten in ZETA möglichen, der hier als Abbruchbedingung verwendet wird. Um die Übersicht zu behalten sollte man sich vielleicht auf den Bereich –10 bis 10 beschränken.

• **15** Dazu definiert man einen neuen Meta-Befehl, `meta__Nachwort`, der während des Spiels nichts macht. Diesen Befehl fängt man dann im Lib-Text ab. Natürlich muss auch die Frage angepasst werden.

```
    Verb 'nachwort' * -> meta__Nachwort:
```

```
Befehl meta__Nachwort {
    Infinitiv "* Nachwort";
    istMeta;
    Aktion = {
        wenn (Ende <= 0) stop "Erst durchspielen,
            dann das Nachwort lesen!";
        "Hast du einmal versucht...
        \n    ... die Hose anzuziehen?
        \n    ... die Tomate zu essen?
        \n    ... die Taschenlampe einzuschalten?
        \n    ...";
    }
}

LibObjekt in ZetaLib {
    Texte = {
    lokal n;
    verschiedenes:
        wenn (n==4 && aBefehl==meta__Nachwort)
            stop Ausf(meta__Nachwort);
        wenn (n==3)
            "Möchtest du einen NEUSTART, einen alten
                Spielstand LADEN, den letzten Zug
                RÜCKGÄNGIG machen";
            wenn (Ende > 1)  ", im NACHWORT
                erfahren, was man im Spiel Lustiges
                machen kann"
            stop " oder das Spiel beENDEn?";
    }
}
```

• **16** Manche Spiele zeichnen eine ASCII-Windrose, hier begnügen wir uns mit einer einfachen Liste:

```
Routine Ritg(i, r, rname) {
    lokal d = Datentyp(aRaum.r);
    wenn (d==Objekt || d==Routine) {
        wenn (i) ", ";
        Text(rname);
        i++;
    }
    antworte i;
}

LibObjekt in ZetaLib {
    Texte = {
        lokal n;
        Status: wenn (n==3) {
            lokal i;
            i = Ritg(i, Norden, "n");
```

```
            i = Ritg(i, Nordosten, "no");
            i = Ritg(i, Osten, "o");
            i = Ritg(i, Südosten, "so");
            i = Ritg(i, Süden, "s");
            i = Ritg(i, Südwesten, "sw");
            i = Ritg(i, Westen, "w");
            i = Ritg(i, Nordwesten, "nw");
            i = Ritg(i, hoch, "h");
            i = Ritg(i, runter, "r");
            i = Ritg(i, rein, "rein");
            i = Ritg(i, raus, "raus");
        }
    }
}
```

Vereinfacht werden hier Routinen als mögliche Ausgänge angenommen. Man kann diese Routinen ja nicht ausführen, da man nicht weiß, ob sie Text ausgeben.

Die lokale Variable `i` wird nur verwendet um zu sehen, ob man ein Komma ausgeben muss. Damit das Hochzählen der lokalen Variable in `Ritg` in der aufrufenden Routine bemerkt wird, muss dieser Wert wieder zurückgegeben werden. Man hätte hier natürlich auch eine globale Variable verwenden können.

• **17** Man definiert für jede Farbe eine Klasse und ein passendes Pluralobjekt, hier als Beispiel nur die blauen Bälle:

```
Klasse blauer_Ball {
    Name = {
        wenn (Liste.Inhalt) stop "blau{e}";
        "klein{e} blau{e} Ball";
    }
    Genus = m;
    Plural = blaue_Bälle;
    Listengruppe = Bälle;
    Adj = 'klein', 'blau';
    Subst = 'ball' m, 'bälle' pp;
}

Objekt blaue_Bälle {
    Name = {
        wenn (Liste.Inhalt) stop "blau{e}";
        "klein{e} blau{e} Bälle{n}";
    }
    Genus = p + Zahlenangabe;
    Listengruppe = Bälle;
    Anzahl;
}
```

Mit der Abfrage nach `Liste.Inhalt` wird geprüft, ob innerhalb einer Unterliste nur ein verkürzter Name ausgegeben werden soll. Nun noch die Listengruppe für alle Bälle:

```
Objekt Bälle {
    Name = {
        "klein{e} Bälle (", Gruppenliste(), ")";
    }
    Genus = p + Zahlenangabe;
    Anzahl;
}
```

• **18** Man definiert einfach zwei Plural-Objekte, so dass eines für die dunklen, ein anderes für die hellen Lampions gilt.

```
Klasse Lampion {
    Name = {
        wenn (selbst ist eingschaltet)
            selbst.Farbe(), " leuchtend";
        sonst " erloschen";
        "{e} Lampion";
    }
    Genus = m;
    Plural = {
        wenn (selbst ist eingeschaltet)
            antworte helle_Lampions;
        antworte dunkle_Lampions;
    }
    Subst = 'lampion' m, 'lampions' pp;
    Nachher = {
        einschalten: mache selbst Lichtquelle;
        ausschalten: mache selbst nicht Lichtquelle;
    }
    Disambig = {
        einschalten:
            wenn (selbst ist nicht Lichtquelle) antworte 8;
        ausschalten:
            wenn (selbst ist Lichtquelle) antworte 8;
    }

    ist schaltbar;
}

Objekt helle_Lampions = {
    Name = "leuchtend{e} Lampions";
    Plural = p + Zahlenangabe;
}

Objekt dunke_Lampions = {
    Name = "erloschen{e} Lampions";
```

```
        Plural = p + Zahlenangabe;
    }
```

Das Wechseln zwischen den Pluralobjekten hätte auch in `Meldung` oder `Nachher` beim an- und ausmachen abgefangen werden können. Dann aber würde nicht berücksichtigt, wenn Lampions nicht vom Spieler angezündet oder ausgeblasen würden, wenn etwa eine Sturmbö das Lampionfest stört.

• **19** Man definiert einfach zwei Plural-Objekte, so dass eines für die dunklen, ein anderes für die hellen Lampions gilt.

```
    Klasse Lampion {
        Name = {
            wenn (selbst ist eingschaltet)
                Text(selbst.Farbwort), " leuchtend";
            sonst " erloschen";
            "{e} Lampion";
        }
        Genus = m;
        Farbe;
        Farbwort = {
            jenach (selbst.Farbe) {
                1:     antworte "gelb";
                2:     antworte "violett";
                sonst: antworte "rot";
            }
        }
        Plural = {
            wenn (selbst istnicht eingeschaltet)
                antworte dunkle_Lampions;
            wenn (selbst.Farbe==1) antworte gelbe_Lampions;
            wenn (selbst.Farbe==2) antworte violette_Lampions;
            antworte rote_Lampions;
        }
        Meldung = {
            ausschalten:
                stop "Du löschst den ", selbst.Farbe(),
                    "en Lampion.";
            einschalten:
                stop "Du zündest den Lampion an, der nun ",
                    selbst.Farbe(), " leuchtet.";
        }
        Wörter = {
            Adjektive:
                wenn (selbst ist eingeschaltet) {
                    wenn (ist_Adjektiv(selbst.Farbwort)) stop;
                    wenn (ist_Adjektiv("hell")) stop;
                    wenn (ist_Adjektiv("leuchtend")) stop;
                } sonst {
```

```
                wenn (ist_Adjektiv("erloschen")) stop;
                wenn (ist_Adjektiv("dunkel")) stop;
            }
        }
        ist schaltbar;
    }

    Objekt heller_Lampion = {
        Name = "leuchtend{e} Lampions";
        Plural = p + Zahlenangabe;
    }

    Objekt dunker_Lampion = {
        Name = "erloschen{e} Lampions";
        Plural = p + Zahlenangabe;
    }
```

Die Lampions haben hier individuelle Farben, die sie, einmal zu einem Plural zusammengefasst, aufgeben. das ist nicht ganz schön, weil der Spieler nun nicht mehr weiß, welche Farben genau die Lampions haben.

Index

"" (Anführungszeichen im Fließtext) **1.9**
--> (Feldzuweisung) **1.8**
-> (bei der Objektdefinition) **2.2**
-- (Dekrement) **1.4**
-> (Feldzuweisung) **1.8**
$ (Kennzeichnung für Hexadezimalzahlen) **1.3**
$ (Option beim Aufruf) **4.1**
$$ (Kennzeichnung für Binärzahlen) **1.3**
+ (additive Eigenschaft) **2.5**
+ (Option beim Aufruf) **4.1**
++ (Inkrement) **1.4**
.# (Anzahl der Einträge einer Eigenschaft) **2.4**
.& (Eigenschaftsfeld) **2.4**
. (Zugriff auf Eigenschaften) **2.4**
... (Bereichsdefinition) **1.6**
/* */ (Kommentar) **1.2**
// (Kommentar) **1.2**
; (zum Trennen von Anweisungen) **1.2**
= (Zuweisungsoperator) **1.3**
== (Vergleichsoperator) **1.6**
[] (Feldindex) **1.8**
\" (Anführungszeichen im Fließtext) **1.9**
\n (Zeilenumbruch im Fließtext) **1.9**
\p (Absatz im Fließtext) **1.9**

-a (Option) **4.3**
abbrechen (Anweisung) **1.7**
Abbruchkriterium **1.7**
#abk (Steueranweisung) **4.3**
Absätze mit \p **1.9**
additive Vererbung **2.5**
aktivesFenster (Systemfunktion) **Anhang A**
Akzente **1.9**
#alias (Steueranweisung) **4.3**
Anführungszeichen **1.3**
Anführungszeichen im Fließtext **1.9**
antworte (Anweisung) **1.5**
Anweisungsblock **1.6**
Argumente **1.5**
Argumente (Systemfunktion) **Anhang A**

`Attribut` (Direktive) **2.3**
Attribute **2.3**, **Anhang A**
Aufrufen von Routinen **1.2**
Aufteilen von Text **3.1**
Ausdrücke **Anhang A**
Ausführungsblock **1.2**
`Ausgabekanal` (Systemfunktion) **Anhang A**

Bedingtes Kompilieren **4.3**
Bedingungen **1.6**, **Anhang A**
`beerbt` (Bedingung) **2.5**
`beerbt` (bei der Objektdefinition) **2.5**
Bereich von Zahlen, gültiger **1.3**
Bezeichner **1.3**, **Anhang A**
Binärzahlen **1.3**
`bis` (Anweisung) **1.7**
Bit **1.4**
bitweise Operatoren
- `^` (exklusiv-Oder) **1.4**
- `&` (Und) **1.4**
- `<<` (Verschiebung nach links) **1.4**
- `>>` (Verschiebung nach rechts) **1.4**
- `|` (Oder) **1.4**
- `~` (Nicht) **1.4**

`#blinddef` (Steueranweisung) **4.3**
`bsBreite` (Systemfunktion) **Anhang A**
`bsHöhe` (Systemfunktion) **Anhang A**

`Char` (Systemfunktion) **Anhang A**
Codeblock **1.6**
`CursorPos` (Systemfunktion) **Anhang A**
`CursorX` (Systemfunktion) **Anhang A**
`CursorY` (Systemfunktion) **Anhang A**

`#datei` (Steueranweisung) **4.3**
`Datentyp` (Systemfunktion) **2.6**, **Anhang A**
Datentypen **2.6**
Dekrement **1.4**
Division durch null **1.4**

`Eigenschaft` (Direktive) **2.4**
Eigenschaften **2.4**, **Anhang A**
Einbinden von Dateien **4.3**

Eingabe von Text **3.1**
`Eingabekanal` (Systemfunktion) **Anhang A**
`#ende` (Steueranweisung) **4.3**
Endlosschleifen **1.7**
`entferne` (Anweisung) **2.2**
`#ersetze` (Steueranweisung) **4.3**
erweiternde Vererbung **2.5**
`erzeuge` (Systemfunktion) **2.8**, **Anhang A**
Erzeugen von Objekten **2.8**

Fallunterscheidung **1.6**
`Feld` (Direktive) **1.8**
Felder **Anhang A**
Felder als Eigenschaften **2.4**
Felder
    Definition **1.8**
    `String` **1.8**
Funktionsaufrufe **Anhang A**

`-g` (Option) **4.1**
geschweifte Klammern **1.2**
globale Bezeichner **Anhang A**
globale Variablen **Anhang A**
Grammatikdefinitionen **Anhang A**
Grundrechenarten **1.4**

`-h` (Option) **4.1**
„Hallo, Welt!“ **1.2**
`hat` (Bedingung) **2.4**
`Hauptprogramm` **1.2**
Hexadezimalzahlen **1.3**
Hochkommas **1.3**

IDs (*identifiers*) **1.3**
implizites `jenach` **10.2**
`in` (Bedingung) **2.2**
`in` (bei der Objektdefinition) **2.2**
Inkrement **1.4**
`ist` (bei der Objektdefinition) **2.3**

`jenach` (Anweisung) **1.6**

`-k` (Option) **4.1**
`Kinder` (Systemfunktion) **2.2**, **Anhang A**
`Klasse` (Datentyp) **2.6**
`Klasse` (Direktive) **2.5**
Klassen **Anhang A**
Klassen von Klassen **2.5**
Klassen von Objekten **2.5**
Kommentar **1.2**
Kommentare **Anhang A**
Kommunikation zwischen Objekten **2.7**
`Konstante` (Direktive) **1.3**
Konstanten **1.3**, **Anhang A**
`kopiereFeld` (Systemfunktion) **Anhang A**

`-l` (Option) **4.1**
Labels **1.7**
`lade` (Systemfunktion) **Anhang A**
`ladeFeld` (Systemfunktion) **Anhang A**
`ladeZug` (Systemfunktion) **Anhang A**
Laufzeitfehler **1.4**
`Lese` (Systemfunktion) **3.1**, **Anhang A**
`leseArg` (Systemfunktion) **Anhang A**
`leseEintrag` (Systemfunktion) **11.2**, **Anhang A**
`#lib` (Steueranweisung) **4.3**
logische Operatoren
`nicht` **1.6**
`oder` **1.6**
`und` **1.6**
`lokal` **1.3**
lokale Bezeichner **Anhang A**
lokale Variablen **Anhang A**
`löscheFeld` (Systemfunktion) **Anhang A**
`löscheFenster` (Systemfunktion) **Anhang A**

`mache` (Anweisung) **2.3**
Mehrfachvererbung **2.5**
messages **2.7**
Mitteilungen **2.7**
Mitteilungen an Objekte **Anhang A**
Morgenstern, Christian **1.9**
`Mutter` (Systemfunktion) **2.2**, **Anhang A**

`num` (Systemfunktion) **1.3**, **Anhang A**

`-o` (Option) **4.1**
`Objekt` (Datentyp) **2.6**
Objektbaum **2.2**, **Anhang A**
Objekte **2.1**
Objektklassen **2.5**
Operatoren **Anhang A**
Operatoren, artihmetische **1.4**
Operatoren
    bitweise **1.4**
    Dekrement- **1.4**
    Inkrement- **1.4**
    Liste **Anhang A**
    logische **1.6**
    Vergleichs- **1.6**
    Vorrang **1.4**
Optionen **4.1**
Optionen
    `-g (große Spieldatei)` **4.1**
    `-h (Hilfe)` **4.1**
    `-k (kritischer Modus)` **4.1**
    `-l (Logbuch)` **4.1**
    `-o (Objekte listen)` **4.1**
    `-s (Statistik)` **4.1**
    `-w (Warnungen unterdrücken)` **4.1**
    `-y (Liste der globalen Symbole)` **4.1**
    `-z (Speichertabelle)` **4.1**

Pfad für einzubindende Dateien **4.1**
`#proto` (Steueranweisung) **4.3**
„Punktrechnung“ **1.4**

Quelltext **1.1**, **Anhang A**

`#release` (Steueranweisung) **4.3**
Release der Spieldatei **4.3**
`Routine` (Datentyp) **2.6**
`Routine` (Direktive) **1.2**
Routinen **1.2**
Routinen
    Argumente **1.5**
    Aufrufen **1.2**, **1.5**
    Bezeichner **1.3**
    Rückgabewert **1.5**

Rückgabewerte **1.5**, **Anhang A**

`-s` (Option) **4.1**
`Schleife` (Anweisung) **1.7**
Schleifen **1.7**, **Anhang A**
schmutzige Tricks
    implizites `jenach` **10.2**
    `leseEintrag` **11.2**
`schreibeArg` (Systemfunktion) **Anhang A**
Schreibweise von Zahlen **1.3**
`Schriftstil` (Systemfunktion) **Anhang A**
`Schwester` (Systemfunktion) **2.2**, **Anhang A**
`selbst` (Systemvariable) **2.7**
Semikolon **1.2**
`Sender` (Systemvariable) **2.7**
`#seriennr` (Steueranweisung) **4.3**
Seriennummer der Spieldatei **4.3**
`solange` (Anweisung) **1.7**
Sonderzeichen **1.9**
`sonst` (Anweisung) **1.6**
`sonst` (Bereich für `jenach`) **1.6**
`#sonst` (Steueranweisung) **4.3**
Speicher für den Compiler **4.1**
`speichere` (Systemfunktion) **Anhang A**
`speichereFeld` (Systemfunktion) **Anhang A**
`speichereZug` (Systemfunktion) **Anhang A**
Spieldatei **1.1**
`Splitte` (Systemfunktion) **3.1**, **Anhang A**
`springe` **1.7**
Sprungmarken **1.7**
`Statusfenster` (Systemfunktion) **Anhang A**
Steueranweisungen **Anhang A**
`stop` (Anweisung) **1.5**, **1.9**
Strichpunkt **1.2**
„Strichrechnung“ **1.4**
`String` (Direktive) **1.8**
Systemfunktionen **1.5**
Systemfunktionen
    `aktivesFenster` **Anhang A**
    `Argumente` **Anhang A**
    `Ausgabekanal` **Anhang A**
    `bsBreite` **Anhang A**
    `bsHöhe` **Anhang A**

Char **Anhang A**
CursorPos **Anhang A**
CursorX **Anhang A**
CursorY **Anhang A**
Datentyp **Anhang A**
Eingabekanal **Anhang A**
erzeuge **2.8**, **Anhang A**
Kind **2.2**
Kinder **2.2**, **Anhang A**
kopiereFeld **Anhang A**
lade **Anhang A**
ladeFeld **Anhang A**
ladeZug **Anhang A**
Lese **3.1**, **Anhang A**
leseArg **Anhang A**
leseEintrag **11.2**, **Anhang A**
löscheFeld **Anhang A**
löscheFenster **Anhang A**
Mutter **2.2**, **Anhang A**
num **1.3**, **Anhang A**
schreibeArg **Anhang A**
Schriftstil **Anhang A**
Schwester **2.2**, **Anhang A**
speichere **Anhang A**
speichereFeld **Anhang A**
speichereZug **Anhang A**
Splitte **3.1**, **Anhang A**
Statusfenster **Anhang A**
Taste **3.1**, **Anhang A**
Text **Anhang A**
Textfarbe **Anhang A**
TextInFeld **Anhang A**
Textumleitung **Anhang A**
Tochter **Anhang A**
Unicode **1.9**
verbleibend **2.8**, **Anhang A**
vernichte **Anhang A**
Vokabel **Anhang A**
Wortumbruch **Anhang A**
zerstöre **2.8**
Zufall **1.5**, **Anhang A**

`Taste` (Systemfunktion) **3.1**, **Anhang A**
`#text` (Steueranweisung) **4.3**
`Text` (Systemfunktion) **Anhang A**
Textausgabe **1.9**
Texte **Anhang A**
Texte komprimieren **4.3**
`Textfarbe` (Systemfunktion) **Anhang A**
`TextInFeld` (Systemfunktion) **Anhang A**
`Textstring` (Datentyp) **2.6**
`Textumleitung` (Systemfunktion) **Anhang A**
`Tochter` (Systemfunktion) **2.2**, **Anhang A**
Typen von Daten **2.6**

Umlaute **1.9**
Unicode **1.9**
`Unicode` (Systemfunktion> **1.9**
Unterklassen **2.5**

`Variable` (Direktive) **1.3**
Variablen
    Bezeichner **1.3**
    globale **1.3**, **Anhang A**
    Gültigkeitsbereich **1.3**
    lokale **1.3**, **Anhang A**
`verbleibend` (Systemfunktion) **2.8**, **Anhang A**
Vergleichsoperatoren **1.6**
Verknüpfungen, logische **1.6**
`vernichte` (Systemfunktion) **Anhang A**
Vernichten von Objekten **2.8**
`verschiebe` (Anweisung) **2.2**
`Vokabel` (Systemfunktion) **Anhang A**
Vorrang von Operatoren **1.4**

`-w` (Option) **4.1**
`weiter` (Anweisung) **1.7**
`wenn` (Anweisung) **1.6**
`#wenn` (Steueranweisung) **4.3**
Werteübergabe **1.5**
white space **1.2**
`wiederhole` (Anweisung) **1.7**
`Wortumbruch` (Systemfunktion) **Anhang A**

`-y` (Option) **4.1**

`-z` (Option) **4.1**
Zahlen in ZETA **1.3**
Z-Code **1.1**
Zeilenumbruch mit `\n` **1.9**
`zerstöre` (Systemfunktion) **2.8**
`Zufall` (Systemfunktion) **1.5**, **Anhang A**
Zufallszahlen **1.5**
Zusammenfassen mehrerer Anweisungen **1.6**
Zuweisen von Werten **1.3**